REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE 2 Dec 97	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Attributes Of Quality Scenarios / Scenario sets Used In Software Requirements Elicitation	5. FUNDING NUMBERS
6. AUTHOR(S) Kimberly Ann Braun	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Colorado at Colorado Springs	8. PERFORMING ORGANIZATION REPORT NUMBER 97-145
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) THE DEPARTMENT OF THE AIR FORCE AFIT/CIA, BLDG 125 2950 P STREET WPAFB OH 45433	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A Approved for public release; Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

DTIC QUALITY INSPECTED 2

14. SUBJECT TERMS	15. NUMBER OF PAGES 147
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Kimberly Ann Braun, Capt (USAF)

M.S., Computer Science 1997

Attributes Of Quality Scenarios / Scenario Sets Used In Software Requirements Elicitation

147 pages

University of Colorado at Colorado Springs

In order for a quality software product to be developed, quality must exist from the beginning. One of the first stages in software development is requirements gathering. Scenarios help bring together the stakeholders of the future system to discuss and agree upon the requirements of the proposed system.

This thesis examines scenarios used in software requirements elicitation. Many different definitions, formats, and ideas exist on scenarios, but no thorough work has been done on what makes a good, quality scenario and scenario set. This thesis will define quality for a scenario and scenario set.

Research into the current state of practice of scenarios will reveal any references authors make with respect to quality attributes they want in their scenarios. Since the result of requirements elicitation is the Software Requirements Specification (SRS), research into what makes a quality SRS will inspire ideas for a quality scenario and scenario set. New, previously unmentioned attributes, generated from fresh, new thinking on the subject will round out the quality attribute list for scenarios and scenario sets that this thesis develops.

Each attribute will be defined, justified, and examples shown of what a scenario or scenario set would be like if the attribute was missing and how the scenario or scenario set would be improved if the attribute were included. Although this paper does not claim to prove the resulting attribute list is sufficient for a quality scenario and scenario set, it will show the necessity of each attribute. Showing how the software lifecycle or other software development functions will be adversely affected if an attribute is missing will prove necessity.

ATTRIBUTES OF QUALITY SCENARIOS / SCENARIO SETS USED IN SOFTWARE REQUIREMENTS ELICITATION

by

KIMBERLY ANN BRAUN

B.A., University of San Diego, 1993

M.S., Colorado Technical University, 1996

A thesis submitted to the Graduate Faculty of the

University of Colorado at Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

1997

This thesis for the Master of Science degree by

Kimberly Ann Braun

has been approved for the

Department of Computer Science

by

[illegible]

Dr. Alan Davis, Chair

[illegible]

Dr. Marijke Augusteijn

Jugal Kumar Kalita

Dr. Jugal Kalita

Nov. 19, 1997

Date

Braun, Kimberly Ann (M.S., Computer Science)

Attributes Of Quality Scenarios / Scenario Sets Used In Software Requirements Elicitation

Thesis directed by Professor Alan M. Davis

In order for a quality software product to be developed, quality must exist from the beginning. One of the first stages in software development is requirements gathering. Scenarios help bring together the stakeholders of the future system to discuss and agree upon the requirements of the proposed system.

This thesis examines scenarios used in software requirements elicitation. Many different definitions, formats, and ideas exist on scenarios, but no thorough work has been done on what makes a good, quality scenario and scenario set. This thesis will define quality for a scenario and scenario set.

Research into the current state of practice of scenarios will reveal any references authors make with respect to quality attributes they want in their scenarios. Since the result of requirements elicitation is the Software Requirements Specification (SRS), research into what makes a quality SRS will inspire ideas for a quality scenario and scenario set. New, previously unmentioned attributes, generated from fresh, new thinking on the subject will round out the quality attribute list for scenarios and scenario sets that this thesis develops.

Each attribute will be defined, justified, and examples shown of what a scenario or scenario set would be like if the attribute was missing and how the scenario or scenario set would be improved if the attribute were included. Although this paper does not claim to prove the resulting attribute list is sufficient for a quality scenario and scenario set, it will prove the necessity of each attribute. Showing how the software lifecycle or other software

development functions will be adversely affected if an attribute is missing will prove necessity.

CONTENTS

CHAPTER

I. INTRODUCTION 1

II. LITERATURE SEARCH RESULTS 6

Definitions 7

Possible Uses Of Scenarios 11

Requirements 11

Design 12

Testing 13

Other Issues 14

State Of The Practice: Scenario Presentation Formats and Ranges Of Media 16

Informal 17

Formal 19

What Makes a Good Scenario? 26

Attributes Of a Quality Software Requirements Specification (SRS) 30

III. APPROACH AND RESULTS 36

Approach 36

Results 39

Definition of a Scenario 39

Goal of Using Scenarios for Requirements Elicitation 39

The Requirements Funnel 40

How to Describe the Attributes 41

Attributes Common to a Quality SRS and Current Thinking of What Makes a Good Scenario 42

1. Complete Scenario 48

2. Complete Scenario Set 49

3. Concise 51

4. Discrete 53

5. Single Scenario Consistency 55

6. Scenario Set Consistency 56

7. Timing Constraints Modeled 58

8. Right Level of Abstraction / Detail 60

9. Understandable 63

Quality Attributes of an SRS not Previously Mentioned for Scenarios, but do Apply to Scenarios 65

10. Achievable 66

11. Appropriate 67

12. Usable after SRS Written {testable / verifiable} 70

13. Named {Traceable} 71

14. Traced 73

Current Quality Attributes of Scenarios not Common with Attributes of a Good SRS .. 75

15. Initial Conditions Described 77

16. Validated 79

17. Single Threaded 82

Quality Scenario Attributes with no Origins 86

18. Model both Normal and Exceptional Cases 87

19. Simulate System Failures and Recoveries as Possible 90

20. Multiple Forms of Media Used as Needed 92

21. Boundaries Between System and Users Shown Clearly 93

Uniqueness and Necessity of Five Similar Attributes: (concise, discrete, single threaded, appropriate, and right level of detail) 95

An Example of a Quality Scenario 101

IV. VALIDATION AND CONCLUSION 104

Requirements 108

Incomplete SRS 108

Incorrect SRS 110

SRS Difficult to Read 111

Other Problems Contributing to a Bad SRS 112

Design and Code 113

Bad Design and Code 114

Difficult to Create a Design 115

Money and Time Wasted 116

Jacobson's Object Use Cases cannot be used 116

Test 116

Don't Know How to Test a Scenario 117

Incorrectly Pass / Fail Test Cases 118

Difficulties in Creating Test Cases 119

Other Problems in Testing 119

Maintenance 120

Correcting the Delivered System 122

Management 124

Mismatch Between Estimated and Actual Cost / Schedule 125

Configuration Management (CM) 127

Inability to put Scenarios under CM Control 127

Conclusion 128

REFERENCE LIST 130

TABLES

Table

1. Scenario Definitions 8
2. Scenario Representations 26
3. Attributes Of a Quality SRS 31
4. Current Thinking on Good Scenario Attributes and Attributes Of a Quality SRS 44
5. If Scenario / Scenario Set Attribute Missing - Resulting Impact on Software Lifecycle 106

FIGURES

Figure

1. Three Worlds of: Quality, Software Requirements Specifications and Scenarios 3
2. My Thesis: Quality Attributes Of Scenarios and Scenario Sets 4
3. Interaction of Actors, Future System and Use Cases 10
4. Jacobson Notation For Combining Use Cases (Taken From Rumbaugh 1994) 10
5. Jacobson's Use Case Model (Taken From Jacobson 1995) 11
6. Jacobson's Use Case Model for Design Showing Software Components that satisfy a Use Case 12
7. Portion of Two-Dimensional Waterfall Model (Davis 1990) showing how Scenarios And Requirements Affect Testing 13
8. Screen 1.0 Welcome Screen For Proposed System 14
9. Screen 1.2 Resulting from User Selecting 'Wildlife' from Welcome Screen 15
10. Screen 1.2.2 Resulting from User Selecting 'Birds' from Screen 1.2 15
11. Diagram of the Monitor Patient Scenario (taken from Kramer and Keng 1988) 20
12. An example of the Basic Model View (taken from Leite et al. 1997) 21
13. An example of the Scenario Model View (taken from Leite et al. 1997) 22
14. Language Extended Lexicon Entry: Picture Cabin (taken from Leite et al. 1997) 23
15. Formal Model of Scenarios - Center for Software Systems Engineering At UCCS (Taken From UCCS 1997) 23
16. Grammar for the Caller View of a Telephone System (taken from Hsia et al. 1994) 24
17. Attributes of a Good Scenario 29

18. Thesis Sources: Current Thinking on Quality Attributes of Scenarios (Quadrants 1 and 3), Quality Attributes of an SRS (Quadrants 1 and 2) and New Thinking (Quadrant 4) 38

19. The Requirements Funnel 40

20. Attributes Common to Current Thinking on Quality Scenarios And SRSs 47

21. SRS Attributes not Common with Current Thinking on Scenario Attributes 65

22. Current Scenario Attributes not Common with SRS Attributes 76

23. New Attributes: Not Found in Current Thinking on Quality in a Scenario Or Quality in an SRS 87

24. Adapted Software Waterfall Lifecycle 104

25. Design and Code Problems Caused by Problems in the SRS Resulting From Missing Scenario / Scenario Set Attributes 114

26. Testing Problems Caused by Problems in the SRS Resulting from Missing Scenario / Scenario Set Attributes 117

27. Source Of Problems in Maintenance Stemmed from Missing Scenario / Scenario Set Attributes 121

28. Possible Causes for Mismatches in Estimated and Actual Cost / Schedule 125

CHAPTER 1

INTRODUCTION

Software consumers are concerned with software quality and want a product that will meet their needs (Keller, Kahn and Panara 1990). Quality is an important, and yet elusive, attribute of software. It is very difficult (if not impossible) to turn an existing software program of poor quality into a program of good quality. Instead, quality needs to be incorporated from the beginning steps of software development. One of the first phases or steps in software development is the 'requirements analysis' phase.

Requirements analysis, as a phase of the software lifecycle, is composed of two main parts: elicitation and specification. Requirements elicitation is the process of gathering requirement information from the stakeholders of a proposed system. Requirements specification is the process of translating the information gathered from elicitation to a specification document from which designers of the system can work to design the system. It is important to elicit and specify as many requirements as possible during the requirements phase because the later in the lifecycle requirements are found to be missing or in error, the more costly it will be to make the corrections (Boehm 1981).

Scenarios can be used as a tool in requirements elicitation. Scenarios are used to show sample interactions with the proposed system. They help users of the future system communicate their wants to developers and help developers communicate how they 'see' the proposed system. Scenarios aid in eliciting requirements that were previously missing. They also help refine and clarify ideas and help get users and developers 'on the same page'.

Scenarios reduce knowledge errors, these are errors caused by not knowing what the true requirements are (A. Davis, Overmyer et al., 1993).

A quality set of scenarios is one that will help elicit and refine/clarify as many requirements as possible. The better the set of scenarios, the better the resulting requirements specification will be, thereby starting down the road to a better quality software product. My thesis will define the attributes of a quality scenario and scenario set.

My thesis will bring together the three worlds of: 1) quality, 2) software requirements specifications (SRS) and 3) scenarios as shown in the Venn diagram in Figure 1. A portion of the scenario world deals with what authors want in their scenarios (translated to quality attributes of their scenarios) and overlaps with the quality world. A portion of the Software Requirements Specification world deals with quality attributes of an SRS and overlaps with the quality world. In addition, some of the attributes of a quality scenario and attributes of a quality SRS are common.

Figure 1 Three Worlds of: Quality, Software Requirements Specifications and Scenarios

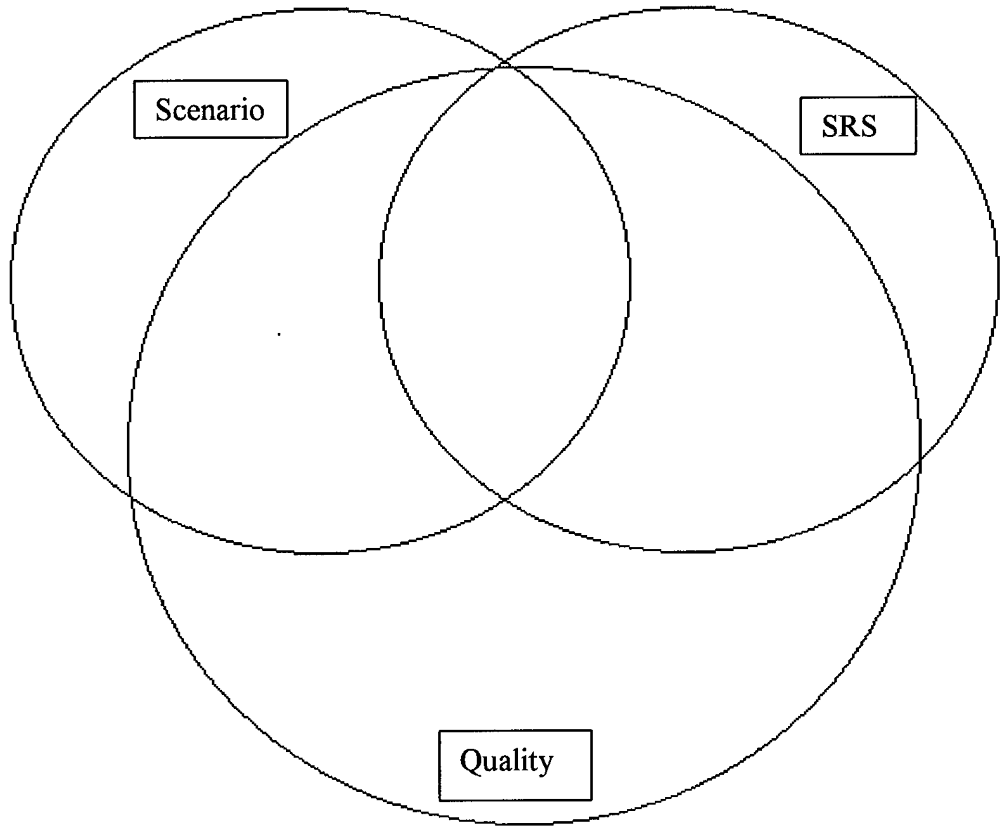
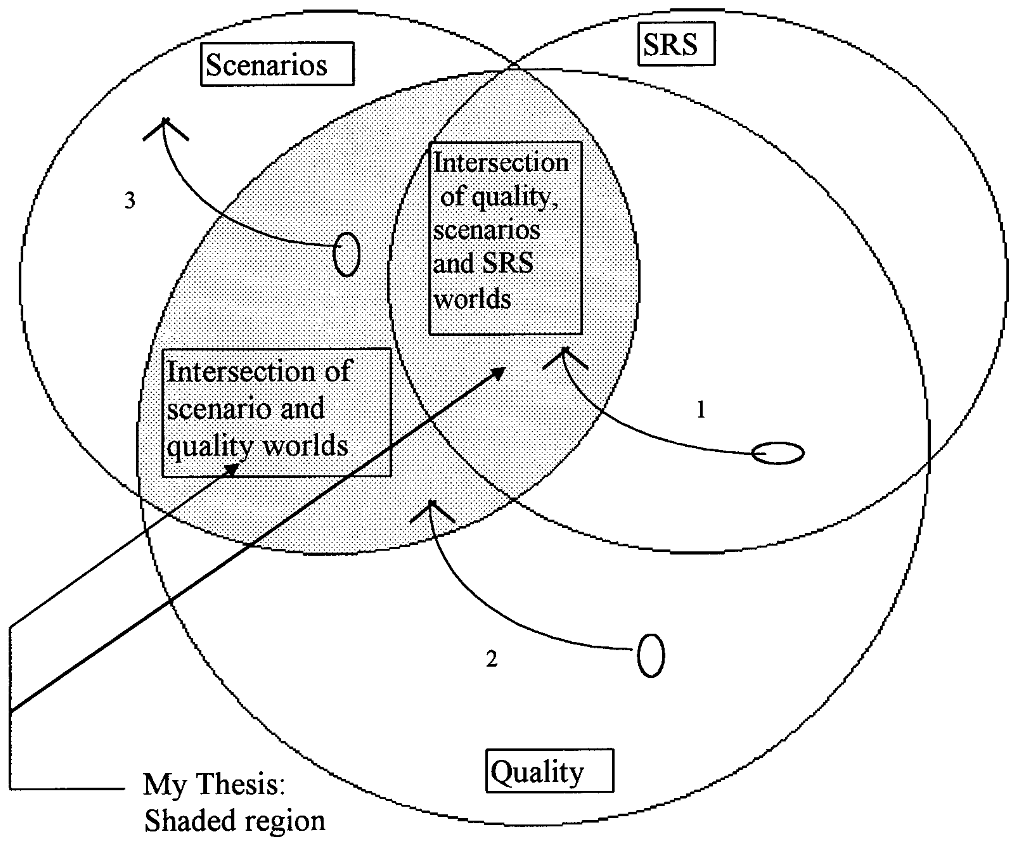


Figure 2 shows how my thesis fits into these three worlds. My thesis covers quality attributes of scenarios and scenario sets and is represented by the shaded region of Figure 2. The shaded area includes quality attributes of scenarios not common to quality attributes of an SRS (the intersection of scenario and quality worlds minus the SRS overlap). In addition, the shaded area includes those attributes of a quality SRS common to attributes of scenarios (the intersection of quality, scenarios and SRS worlds). My thesis will expand the shaded region.

Figure 2 My Thesis: Quality Attributes of Scenarios and Scenario Sets


Scenarios
SRS
Intersection of quality, scenarios and SRS worlds
3
Intersection of scenario and quality worlds
1
2
My Thesis: Shaded region
Quality

My thesis will expand the shaded region by claiming there are some attributes of an SRS not currently belonging to the intersection with scenarios that are also good for scenarios (as shown by arrow #1 from the 'SRS' world to the intersection of quality, scenarios and SRS worlds). In addition, I claim there are also attributes in the 'quality' world not previously mentioned that pertain to scenarios (as shown by arrow #2 from the 'quality' world to the intersection of scenario and quality worlds). Some authors have discussed quality attributes of their scenarios that they desire (interpreted as quality attributes of a scenario). While most of these apply to all scenarios, I claim that some do not (as shown by arrow #3 leaving the intersection of scenario and quality worlds to the non-intersecting portion of the scenario world).

To support these claims I will list, define and justify a list of attributes that make up a quality scenario and scenario set. The list of attributes will be created by 1) expanding upon and adapting quality attributes previously mentioned by authors on scenarios and SRSs and 2) adding to the list of attributes new attributes not previously referenced in the realm of SRSs or scenarios. In addition, I will discuss why some authors' current beliefs on quality scenarios are not applicable to all scenarios. I will prove that each attribute in my list is necessary by showing how the software lifecycle and other software functions would be adversely affected if any one of the attributes was missing. While I will prove necessity, I am not claiming sufficiency. Even though countless hours have been spent pouring over the attribute list and thinking of all possible situations and attributes, I cannot prove that there is not one attribute lingering 'out there' that has not been included, yet would aid in making a scenario or set of scenarios better (although I cannot think of any).

By accomplishing the above, I will be adding new information to the world of scenarios for requirements elicitation. In total, I will define 21 attributes of a quality scenario / scenario set. Previously, no more than 5 scenario attributes have been mentioned at one time (by Nardi in 1995).

CHAPTER 2

LITERATURE SEARCH RESULTS

Scenarios are not a well-specified technology. It is difficult to find two people with the same definition of a scenario with respect to software systems. Not only is there a range of definitions for scenarios, but the degree of informality/formality, forms of media, and the range of uses of scenarios vary greatly. There is no consensus on the appropriate level of detail for a scenario (Rosson and Carroll 1995).

The fact that thoughts and ideas vary greatly in the world of scenarios is fairly evident by just looking at the titles of some articles written in the 1992 SIGCHI Bulletin:

- Will The Real Scenario Please Stand Up? (Campbell 1992),
- What's in a Scenario? (Wright 1992),
- Multiple Uses of Scenarios (Young and Barnard 1992),
- Scenario? Guilty! (Kyng 1992)

With all the research and work into scenarios, there has yet to be created a thorough list of attributes that a good quality scenario should possess and how to check that a scenario contains those attributes. However, research has been done into what makes a quality Software Requirements Specification (SRS) (the end product that elicitation scenarios help produce). This chapter will not only look at work on scenarios, but will also look at the work in describing and measuring quality in an SRS.

Definitions

As already mentioned, there is a great variety of definitions for scenarios in software systems. Table 1 is a selection of different definitions for scenarios. The definitions differ in how specific or constraining they are. Some definitions of a scenario are general or wide open such as: ‘an evolving description of situations in the environment’ (Leite et al. 1997) or ‘a story that illustrates how a perceived system will satisfy a user’s needs’ (Holbrook 1990) or ‘a flexible, informal medium for carrying on a high-level conversation between groups of designers and users’ (Wexelblat 1987). Other definitions are more specific such as: ‘a path in a scenario tree described by a grammar from which a conceptual machine is constructed’ (Hsia et al. 1994) or ‘a sequence of actions showing how a transition from one state to another might occur’ (Anderson and Durney 1992). The more specific definitions seem to dictate a methodology such as using state transitions or constructing scenario trees and grammars while the more general definitions give very little guidance for a methodology.

Table 1 Scenario Definitions

A scenario is......	Source
simply a proposed specific use of the system… a scenario is a description of one or more end-to-end transactions involving the required system and its environment	(Potts, Takahashi, Anton 1994)
an encapsulated (that is self-contained, portable) description of: • An individual user • Interacting with a specific set of computer facilities • To achieve a specific outcome • Under certain circumstances • Over a certain time interval (this is in contrast to simple static collections of screens and menus; the scenario explicitly includes a time dimension of what happens)	(Nielsen 1995)
a narrative that describes someone trying to do something in some environment. As such it is a description of context, which contains information about users, tasks and environment	(Karat 1995)
an informal narrative, collaboratively constructed by a team, that describe human work processes; some of these work processes involve a computer….we attempt to address, where appropriate, human goals and motivations, design alternatives, and other questions that extend or deepen our understanding of the problem…	(Muller et al. 1995)
a narrative format, as in text narrative or storyboard or video which involves the inclusion of user content	(Nardi 1995)
a story that illustrates how a perceived system will satisfy a user's needs	(Holbrook 1990)
a path in a 'scenario tree' described by a grammar from which a conceptual machine is constructed	(Hsia et al. 1994)
a simulation of events a user would experience in performing the tasks that constitute the operation of a system	(Hooper and Hsia 1982)
an evolving description of situations in the environment	(Leite et al. 1997)
a sequence of actions showing how a transition from one state to another might occur	(Anderson and Durney 1992)
a flexible, informal medium for carrying on a high-level conversation between groups of designers and users	(Wexelblat 1987)
a sequence of interactions between a solution system and its environment which serves as a representative example of how some system features work	(UCCS 1997)

On a similar note, a 'use case' is	Source
a way to use the system while a scenario is an instance of a use case	(Jacobson 1995)
a description of the possible sequences of interactions among the system and one or more actors in response to some stimulus by one of the actors. It is not a single scenario, but rather a description of a set of potential scenarios, each starting with some initial event from an actor to the system and following the ensuing transaction to its logical conclusion	(Rumbaugh 1994)

Use cases and scenarios may have slightly different means to reach the same end. That is why use cases are briefly explained here along with other authors' views of scenarios.

The boundary between scenarios and use cases is almost a blur depending on one's definition of a scenario. Table 1 shows two definitions of use cases. According to two scholars, James Rumbaugh and Ivar Jacobson, a use case is a description of a set of potential scenarios. Therefore, a scenario is a specific instance of a use case. Rumbaugh and Jacobson are particularly interested in how use cases can lend themselves to object-oriented modeling. For example, a use case is a description for a set of scenarios, in the same sense that a class is a description for a set of objects (Rumbaugh 1994).

When working with use cases, an actor (i.e. system operator, database administrator, separate computer, etc.) is defined as any outside entity that interacts with the system and each actor uses the system in fundamentally different ways. Any person may have many different actor roles (i.e. database administrator, user, maintainer, etc.) Each of these different ways that an actor interacts with the system is a use case. Figure 3 illustrates this relationship among actors, the system and use cases.

Use cases are written in natural language and can be combined to create other use cases (Rumbaugh 1994). In addition to natural language, diagrams can also be used to 1) show the relation among use cases (which use cases 'uses' or 'extends' other use cases) as shown in Figure 4, and 2) to show the interaction that takes place when objects send stimuli to one another (interaction diagrams). Jacobson has a method for modeling the entire proposed system using use cases. This model is a graph with actor nodes (aX), use case nodes (uX), and communication arcs as shown in Figure 5. The actor nodes represent the actors, the use case nodes represent a natural language written use case, and communication arcs show which actors deal with which use cases (Jacobson 1995).

Figure 3 Interaction of Actors, Future System and Use Cases

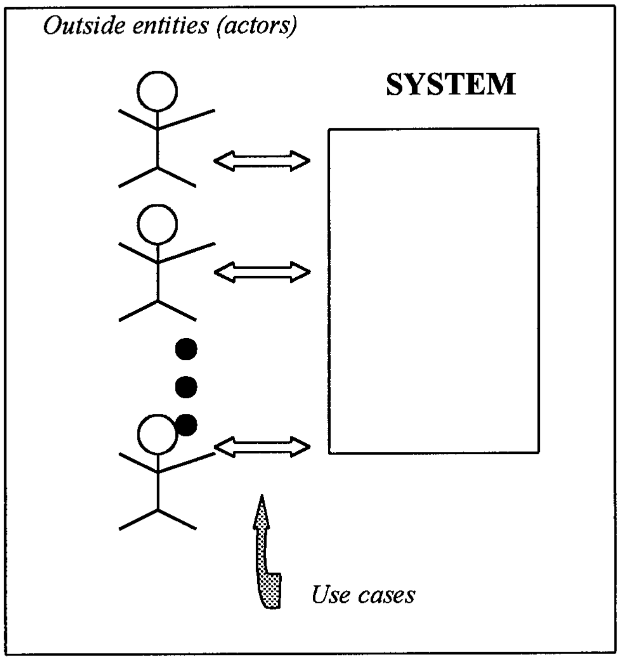


Figure 4 Jacobson Notation for Combining Use Cases (taken from Rumbaugh 1994)

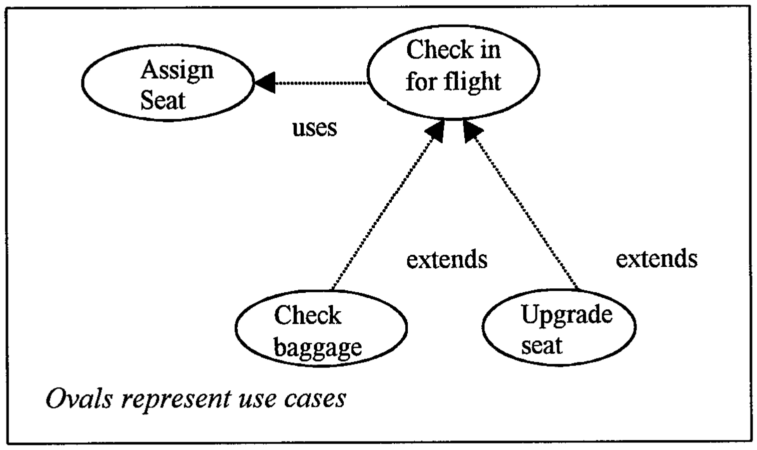
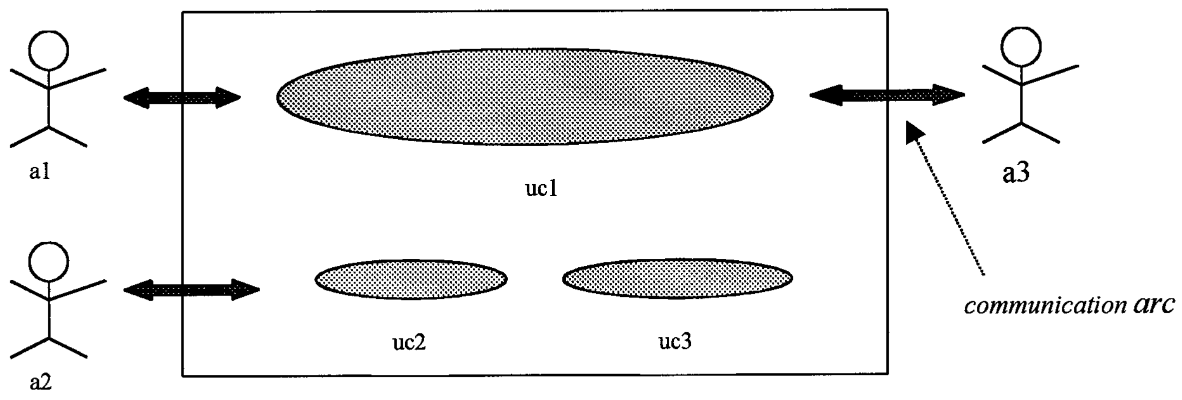


Figure 5 Jacobson's Use Case Model (taken from Jacobson 1995)



Possible Uses Of Scenarios

Scenarios for software systems have many uses throughout the entire software lifecycle. Scenarios can aid in requirements elicitation and analysis, design, testing and so on.

Requirements

Scenarios help elicit and clarify requirements. Scenarios are effective in prompting and answering questions about requirements. They simulate future situations with the target system and allow end users to experience, to some degree, what it would be like to work with the future system. They allow users to evaluate and comment on the suitability of the proposed system. Scenarios help draw out the non-explicit knowledge and experience that users and other stakeholders possess which help build up and clarify requirements. Scenarios also help bring to light the relation between functional and non-functional requirements.

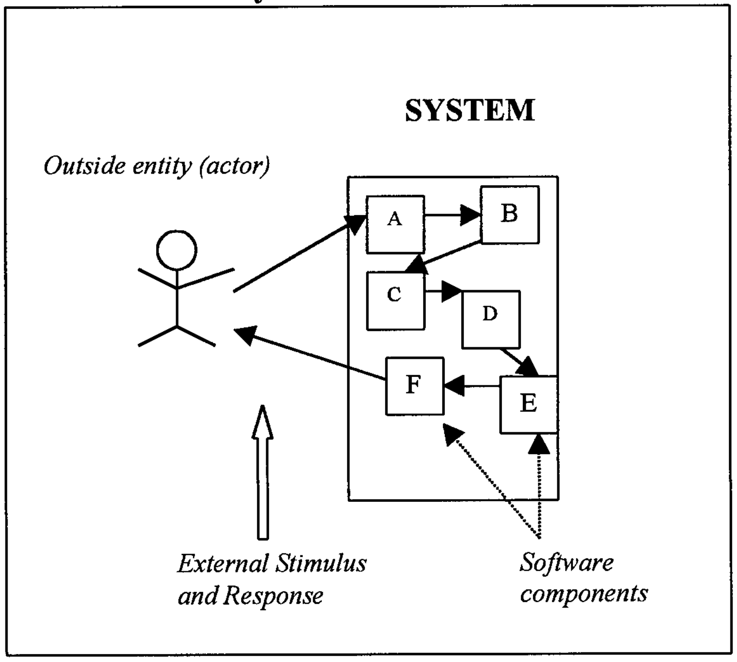
(Potts, Takahashi, Anton 1994; Carroll 1995; Kyng 1995; Karat 1995; Holbrook 1990; Leite et al. 1997)

Quite often, working with one scenario will stimulate ideas about other scenarios or situations, leading to more robust requirements. Scenarios also help validate the requirements specification (Kramer and Keng 1988; Hsia et al. 1994).

Design

Ivar Jacobson uses object use cases in the design phase to create object models. Object use cases show the interaction among software components as a result of an external stimulus as shown in Figure 6. An actor's input to the system, as shown in a use case, is the external stimulus. For each use case, software objects that will accomplish the use case are identified and described. (Jacobson 1995)

Figure 6 Jacobson's Use Case Model for Design Showing Software Components that satisfy a Use Case

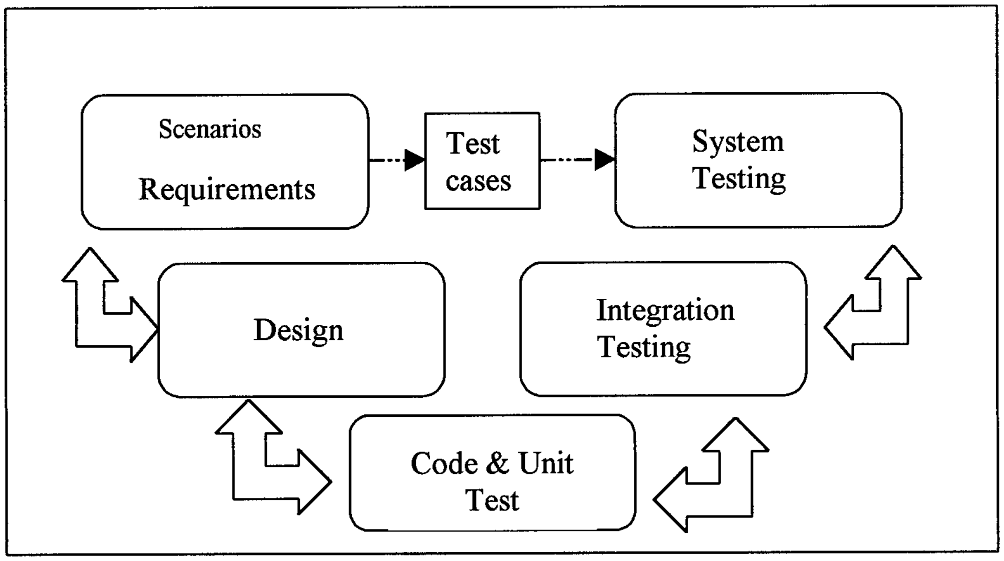


In addition to using use cases in design, the behavior of the design can be communicated to users via scenarios and the suitability of the design evaluated (and corrected) before the final system is delivered. Scenarios are useful tools to facilitate communication between users and designers. Watching users and other stakeholders work with scenarios help designers observe behaviors and know why the behaviors were chosen and also prompts users to indicate the purpose and constraint expected of each action or step in a scenario. (Nardi 1995; Holbrook, 1990; Sutcliffe 1997)

Testing

Scenarios provide a basis for testing also. Scenarios drive requirements. Together, scenarios and requirements drive test cases for system testing as shown in Figure 7. If the final system does not behave as described in an approved scenario, then the system may not be acceptable (and vice versa). (Hsia et al. 1994; Nielsen 1995)

Figure 7 Portion of Two-dimensional Waterfall Model (Davis 1990) showing how Scenarios and Requirements Affect Testing



Other Issues

There is a multitude of other uses, which do not fall into one exact phase of the software lifecycle. The field of HCI (Human Computer Interaction) benefits from being able to use scenarios to first run through proposed screens and operations to help ensure goals are being met and to test HCI theories. Figure 8, Figure 9, and Figure 10 show a fictitious sequence of such screens. Users and other stakeholders can be shown these screens and the actions taken to get to each screen and evaluate the suitability of the proposed design. User-centered design and participatory design, where the end user is explicitly involved in the design process, benefits greatly from the use of scenarios. Scenarios facilitate user and designer interaction and involvement and help each side get their views of the system across. (Campbell 1992; Nielsen 1995; Chin, Rosson and Carroll 1997)

Figure 8 Screen 1.0 Welcome Screen for Proposed System

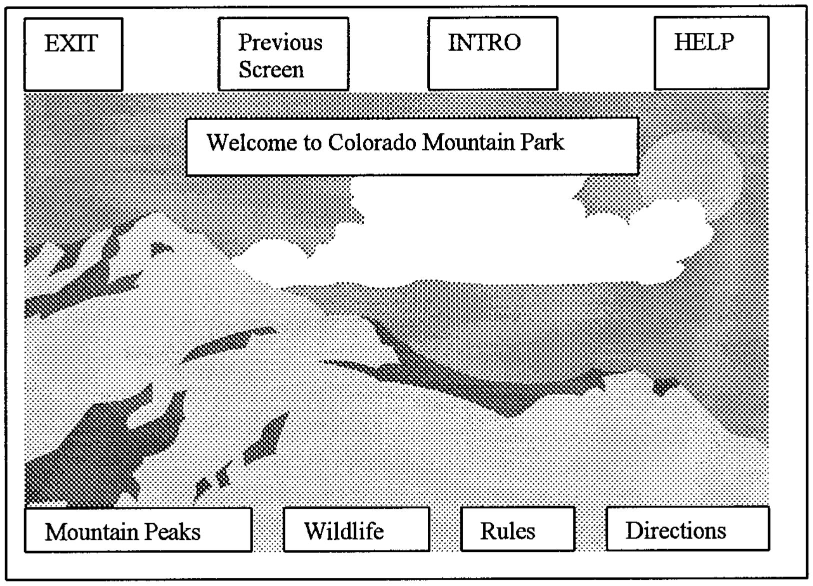


Figure 9 Screen 1.2 Resulting from User Selecting 'Wildlife' from Welcome Screen

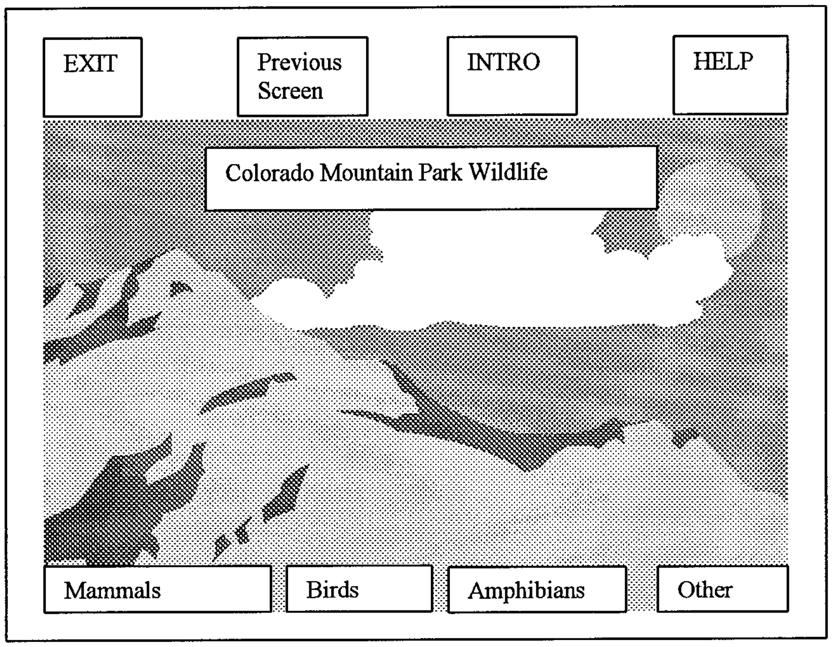
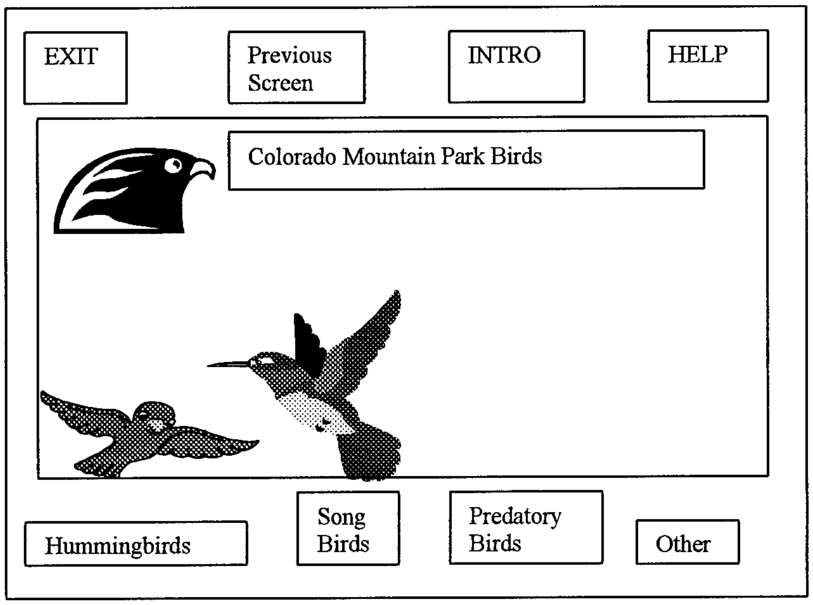


Figure 10 Screen 1.2.2 Resulting from User Selecting 'Birds' from Screen 1.2



Scenarios help clarify policy issues and division of responsibility between the system and user along with challenging assumptions about system boundaries. As Muller et al. point out, scenarios are good at 'problematizing' situations. Problematizing is the process of transforming one's assumptions (that may be assumed differently by different people) into open questions. They help bring unexamined, tacit knowledge out to the open. (Potts, Takahashi, Anton 1994; Muller et al. 1995)

Scenarios provide a basis for training and team building. Many times scenarios are very useful to describe in users manuals and other documentation. (Karat 1995; Muller et al. 1995; Kramer and Keng 1988)

State Of The Practice: Scenario Presentation Formats and Ranges Of Media

Scenarios can take on nearly any form possible. Jack Carroll states some of those forms to be: textual, storyboard, video mockup, scripted prototype or even a physical situation contrived to support certain physical activities (Carroll 1995). However, many authors have gone beyond those mentioned by Carroll, broadening the range to include nearly everything from an informal text or drawing to a more formal grammar, Language Extended Lexicon, or animation of data flows (Hsia et al. 1994; Leite et al. 1997; Kramer and Keng 1988; Muller et al. 1995). There are authors who advocate an informal representation and there are authors who advocate a more formal or structured representation. For the purpose of this paper, informal methods are those which do not take a lot of preplanning nor have strict rules to follow for creating a scenario while formal methods do take planning (i.e. animation/prototyping) or have more rules or guidelines for creating a scenario (i.e. scenario trees and use cases). The boundary between informal and formal is blurry and not very important for this thesis.

Informal

Joseph Goguen, who was with the Centre for Requirements and Foundations at Oxford University, claims that the requirements process is social in nature. Therefore, requirements engineering can never be an entirely formal process because the goal of requirements elicitation and analysis is to discover stakeholders' needs and reconcile them with technical possibilities (Goguen 1993). Tom DeMarco shares this opinion (DeMarco 1996).

Muller et al. are proponents of using a low-tech technique, claiming that using low tech representations help team members maintain a focus on the users' work process (perhaps by spending less time focusing on the 'gadget' the scenario is represented by or the medium's capabilities which are irrelevant to the scenario). In addition, a low-tech representation helps the team break out of an unsuccessful design. Their process involves the use of cards and other office supplies (such as Post-Its, highlighters, colored paper, etc.) to simulate workflows and ideas about design. These materials can be easily manipulated and changed. (Muller et al. 1995)

One of the most informal representations of a scenario is natural language (e.g. English text). The next chapter of this thesis gives examples of written scenarios. Authors such as Karat and Macaulay use plain text representation in their elicitation techniques. Linda Macaulay brings in all stakeholders to aid in requirements elicitation and requires that the language and terminology used in the cooperative activity of requirements elicitation be readily understandable by all stakeholders. Stakeholders are all of those who have a stake in the change being considered, those who stand to gain and stand to lose (Macaulay 1992). Chin, Rosson and Carroll stress that terminology must be familiar to all participants and that the terminology, which emerges, relies on the language of the user (Chin, Rosson and Carroll 1997). Karat, too, believes that the problem should be expressed in an easy-to-understand text description. He used scenarios in the design of a speech recognition system (Karat 1995).

Another means of creating scenarios is described by Jakob Nielsen as a 'diary scenario'. A diary scenario is simply a user writing down the activities and situations they encounter during the day which are relevant to the system. The diary scenario differs from most scenarios because it describes real observations rather than ideas about a non-existent system that will be built in the future (Nielsen 1995).

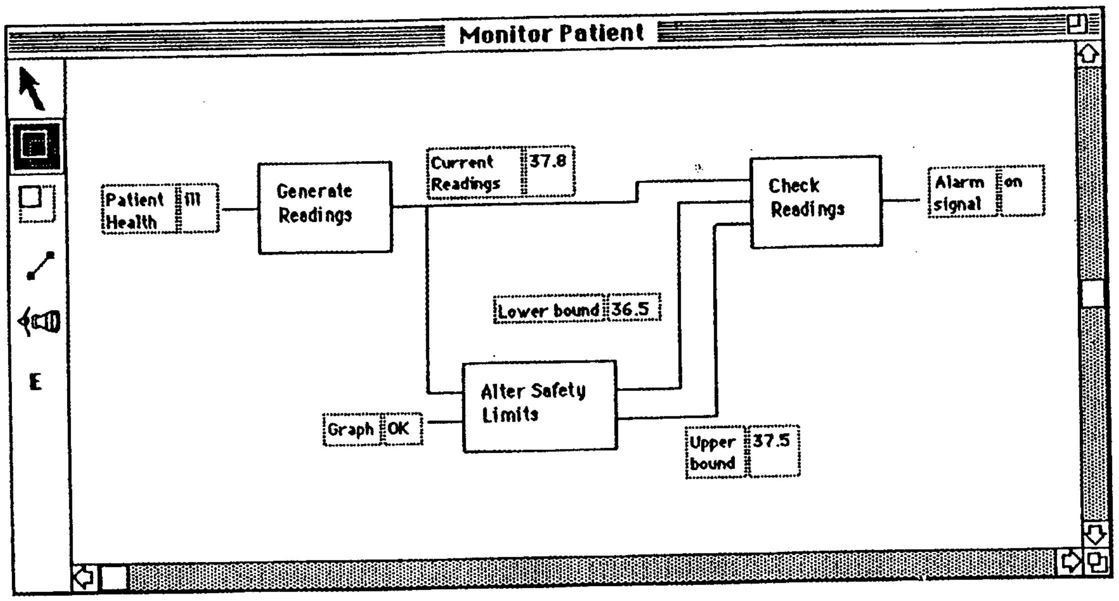
Storyboarding (a technique originally created by Disney in the 1930s (Zahniser 1993)) can span the range between informal and more formal scenarios. A storyboard is a sequence of displays that represent functions that the system may perform when implemented (Andriole 1989) as shown in Figure 8, Figure 9, and Figure 10. This may be as simple as using paper and pencil (or other office supplies as Muller et al. have done) to sketch out possible screens or workflows, or the storyboard may be created using a computer and even animated. Storyboarding is a popular technique because it is cost-effective while being a dynamic and 'live' tool. It also provides a means to test alternatives quickly (Andriole 1989).

Formal

A scenario may be represented via a prototype. Thomas Erickson's definition of a prototype includes anything from a pencil sketch or foam mock up to a slideshow, videotape, or partial implementation (Erickson 1995). Arguably, a pencil sketch or foam mock-up could be thought of as an informal method while partial implementations thought of as more formal. Generally, when dealing with scenarios, a prototype will be created to simulate that particular scenario or path through the system and may not be robust enough to show any other functionality. An example of such a prototype is Alistair Sutcliffe's 'concept demonstrator' which is a limited prototype to run a scenario (Sutcliffe 1997). Sutcliffe used the concept demonstrator for a shipboard emergency management system. Hooper and Hsia also advocate the use of a 'quick and dirty' prototype to run their scenarios. They provide a sketch of the system based on the requirements as perceived. Their attempt is to capture the conceptual system as visualized by the user by use of operational examples or scenarios. They do not find it necessary to model the system or any component directly, but rather represent the performance of the system for selected events (Hooper and Hsia 1982).

Several authors represent their scenarios using diagramming or data flow techniques. Animating the data flows is one way of playing out a scenario. Animation adds a dynamic or real time feel to the scenario. When input is needed from the user, or when there is a part of the scenario for which there is missing information, dynamic prompts can ask users for information as the scenario is running (Kramer and Keng 1988). Figure 11 shows a diagram of one Kramer and Keng's scenarios. When animated, the user is guided along the diagram showing them where in the scenario they are currently.

Figure 11 Diagram of the Monitor Patient Scenario (taken from Kramer and Keng 1988)

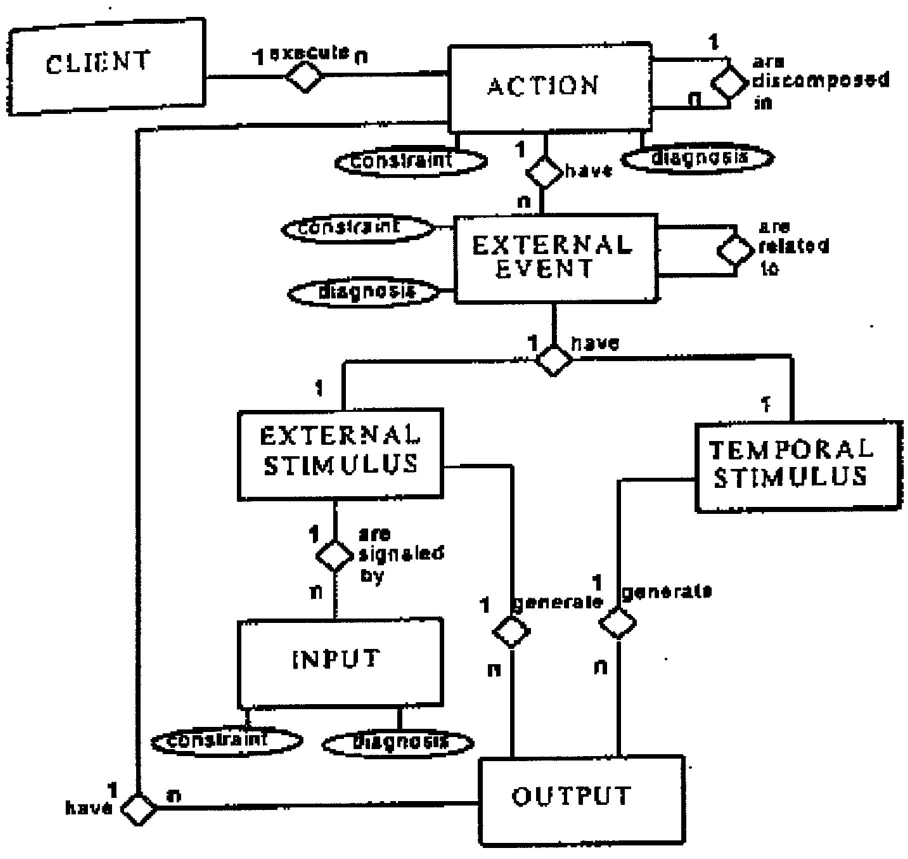


Graphical description of a transaction

Use cases combine written English with diagramming for an overall model as described on page 9 , blending the line between being an informal or formal method. The written use case describes an actor's interaction with the system in a specific case and interaction diagrams show how the objects, events and stimuli interact. Another notation shows how the use cases themselves fit together (i.e. which use case 'uses' or 'extends' another use case) as shown in Figure 4 (Rumbaugh 1994; Jacobson 1995).

Leite et al. combine many different diagramming techniques to create different views of a scenario to make up their 'Requirements Baseline Conceptual Model'. The Basic Model View and Scenario Model View use the entity relation framework or diagrams (ER diagrams) with examples shown in Figure 12 and Figure 13. The Lexicon View is composed of a Language Extended Lexicon that records the signs, words or phrases that are peculiar to the domain and is not a diagram itself (similar to a dictionary). Figure 14 is an example of a Language Extended Lexicon entry based on the passport emission domain. Different views of a scenario can be seen using hypertext. (Leite et al. 1997)

Figure 12 An example of the Basic Model View (taken from Leite et al. 1997)



The ER Diagram for the Baseline Basic Model

Figure 13 An example of the Scenario Model View (taken from Leite et al. 1997)

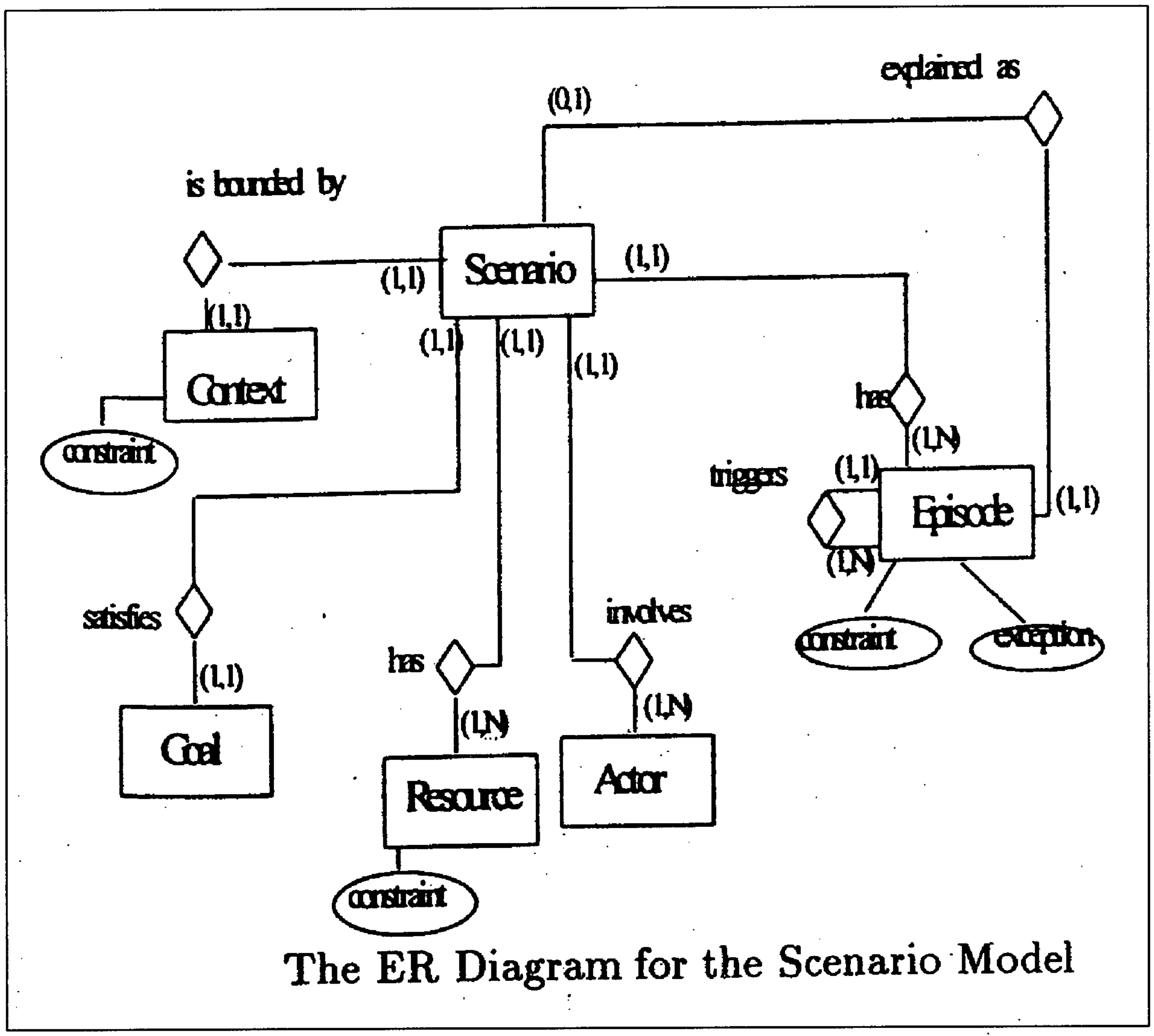


Figure 14 Language Extended Lexicon Entry: Picture Cabin (taken from Leite et al. 1997)

Picture Cabin

- Notion:
  - It is a sector of the Documents and Certificates Division
  - It is where the citizen's picture is taken and charged
- Behavioral Response:
  - The form is stamped with the same number as the picture
  - The citizen receives two pictures
  - The picture cabin clerk archives the third picture

The Center for Software Systems Engineering at the University of Colorado at Colorado Springs uses the Backus Naur Form (Backus 1959) grammar to formally define its scenarios (see Figure 15). Its scenarios have four basic elements: inputs to the system, outputs from the system, timing constraints and set of initial conditions. (UCCS 1997)

Figure 15 Formal Model of Scenarios - Center for Software Systems Engineering at UCCS (taken from UCCS 1997)

SCENARIO	= <IC> <SCENARIO BODY>
SCENARIO BODY	= <SCENARIO STEP> | <SCENARIO BODY> <SCENARIO STEP>
SCENARIO STEP	= <INPUT> <I-O>^{P} <OUTPUT> [<O-I>]^P
INPUT	= <I>^{M}
I	= External-entity Action [<I-I>^{R}]
OUTPUT	= <O>^{N}
O	= Action External-entity [<O-O>^{R}]
IC	= Initial states and conditions
I-I	= A maximum time allowed between two stimuli*
I-O	= A maximum time allowed between the arrival of the stimuli and the system's response*
O-I	= A maximum time allowed between the system's response and the next stimulus from the environment*
O-O	= A maximum time allowed between 2 or more system responses*

Superscripts indicate item can occur 1 to M,N,P or R times * = Dasarathy 1985

Another formal approach to creating scenarios is to combine scenario trees, a grammar and a conceptual (finite state) machine as detailed by Hsia et al. The scenario tree is composed of nodes with each node representing a state as the user perceives it. The initial state is the root node. The tree is then converted into a formal grammar such as the one in Figure 16 using an algorithm. A deterministic finite state machine is then created from the grammar. This finite state machine is called a conceptual state machine. Because of the algorithm and formal method used, Hsia et al. claims this method ensures consistency, lack of redundancy, and internal completeness of the generated scenarios. (Hsia et al. 1994)

Figure 16 Grammar for the Caller view of a Telephone System (taken from Hsia et al. 1994)

G_v = (NT, S, R, A)

S = {Off_H. Not9, d, R, Cpu, talk, Chup, On_H}

NT= {Caller, <FD>, <Int>, <Third>, <Fourth>, <Action>, <Talk>, <Fin>}

A= Caller

R= {Caller -> Off_H <DialTone>,
<DialTone> -> Not9<InternalCall>,
<Int> -> digit<Third> | On-H
<Third> -> digit<Fourth> | On-H, <Fourth> -> digit<Validating> | On-H,
<Validating> ->Ring<Connecting> | On_H | Busy<TryAgain> | <disconnected>

John Anderson and Brian Durney take a different approach. They use scenarios to identify missing capabilities that, if included, would enable system users to reach their goals. They also use scenarios to determine whether a particular objectives set will allow prohibited transactions. They use as input a set of supported and prevented objectives expressed as transitions between states. Their approach searches for incompleteness (not being able to reach a goal) and unsafeness (being able to complete an action that should not be allowed).

Chin, Rosson and Carroll supplement written scenarios with videotaped scenarios as part of their Task Artifact Framework (TAF). TAF has four stages: scenario generation, claims analysis, features envisionment and scenario envisionment. They find that written scenarios tend to be a bit more abstract and less real than video scenarios. With video scenarios, participants can see the scenario in the same form as they experience it (Chin, Rosson and Carroll 1997). Depending on the amount of preplanning needed for a video scenario, it could be argued that video scenarios are informal scenarios.

The range of representations of scenarios varies greatly. Table 2 shows a brief synopsis of the formats discussed.

Table 2 Scenario Representations

"Informal"	
Cards and other office supplies ➢ (Muller et al. 1995)	Natural language (i.e. written English) ➢ (Karat 1995; Macaulay 1992; Chin, Rosson and Carroll 1997)
Diary ➢ (Neilsen 1995)	Storyboarding (paper) ➢ (Muller et al. 1995; Rosson and Carroll 1995)
Physical situations contrived to support user activities ➢ (Carroll 1995)	
"Formal"	
Prototype ➢ (Sutcliffe 1997; Hooper and Hsia 1982; Carroll 1995; Erickson 1995; Nielsen 1995)	Animation of data flows ➢ (Kramer and Keng 1988)
Use cases ➢ (Jacobson 1995; Rumbaugh 1994)	Requirements baseline conceptual model (Entity Relation diagrams, Lexicon, hypertext) ➢ (Leite et al. 1997)
Scenario tree, grammar and conceptual state machine ➢ (Hsia et al. 1994)	Video mock-ups / scenarios ➢ (Carroll 1995; Chin, Rosson and Carroll 1997)
Storyboarding (computer) ➢ (Andriole 1989)	Supported and prohibited objectives to test for unsafeness and incompleteness ➢ (Anderson and Durney 1992)
Backus Naur Form grammar ➢ (UCCS 1997)	

What Makes a Good Scenario?

Not a lot of literature exists on what makes a good scenario (Karat 1995), but some authors have alluded to it. For example, Holbrook states that tutorials in users manuals are good examples of scenarios (although he does not state what it is about the tutorials that make them good examples) (Holbrook 1990). Nielsen states that scenarios should explicitly include a time dimension of what happens (Nielsen 1995) and, along the same lines, Kramer and Keng discuss the need to model timing behavior in animated scenarios (Kramer and Keng 1988).

Hsia et al.'s method allows for the checking of correct, complete, consistent and validated scenarios (Hsia et al. 1994). The Center for Software Systems Engineering at the University of Colorado at Colorado Springs strives for scenarios that are complete, consistent, correct, and with initial conditions described (UCCS 1997).

Many people believe that scenarios should reflect what the users are or will be doing. Scenarios must be described in the natural work setting and grounded in activities of the real world in order to get 'buy-in' from the users. (Nardi 1995; Chin, Rosson and Carroll 1997) In the same vein, use cases, like scenarios, must strive to solve the right problem by involving the users in analysis (Rumbaugh 1994).

Some people claim that concrete, specific scenarios are better than general or ambiguous scenarios. They claim concreteness is an important attribute because concrete scenarios describe particular instances of use and users work in the specific and not the abstract. Concrete scenarios help surface atypical events, while ambiguous scenarios may gloss over the exceptional cases and stick to the typical situations. (Potts, Takahashi, Anton 1994; Carroll 1995; Kyng 1995; Rosson and Carroll 1995)

However, in the beginning stages of requirements gathering, Thomas Erickson likes scenarios that may be a bit more ambiguous because of their ability to gather more information. Like stories, ambiguous scenarios have many interpretations leading to people swapping stories or scenarios and aiding in team building. People will fill in the gaps of ambiguous scenarios differently, again leading to different sources of ideas. A 'rough' or incomplete scenario gives the feeling that it 'ain't done yet' (Erickson 1995) and decreases the level of commitment to design. When users think that a lot of time, thought and effort have gone into a design, they may tend to limit their ideas or thoughts to those similar to the design shown. A less polished scenario allows people to be freer in coming up with ideas instead of thinking they are tied down to something similar to the design shown through the scenario.

Whether concrete or abstract, scenarios must be understood by all participants (Chin, Rosson and Carroll 1997, Karat 1995). The language and terminology used in the scenarios should be understandable by all stakeholders involved in requirements elicitation. If they cannot understand the scenario, stakeholders will not be able to clearly analyze the scenario.

Morten Kyng prefers requirements scenarios that are closed with no external references and reflect situations that the software system will support. They serve as discussion tools for software design, and not the work situation (Kyng 1995). In other words, after the entire workflow has been described, requirements scenarios should only describe the software system interactions.

As far as length is concerned, Nardi believes that good scenarios should be short, fun and vivid. He also believes that maintaining data quality is important (Nardi 1995). With respect to use cases, James Rumbaugh claims a use case 'follows a [single] thread of control in and out of the system' (Rumbaugh 1994). Figure 17 highlights the different attributes that are thought to make a good scenario.

Figure 17 Attributes of a Good Scenario

- Timing behavior modeled
  - (Nielsen 1995, Kramer and Keng 1988)
- Correct
  - (Hsia et al. 1994, UCCS 1997)
- Consistent
  - (Hsia et al. 1994, UCCS 1997)
- Complete
  - (Hsia et al. 1994, UCCS 1997)
- Initial Conditions Described
  - (UCCS 1997)
- Incomplete
  - (Erickson 1995)
- Validated
  - (Hsia et al. 1994)
- Concrete / Specific
  - (Potts, Takahashi, Anton 1994; Carroll 1995; Kyng 1994; Rosson and Carroll 1995)
- Ambiguous / Rough in beginning
  - (Erickson 1995)
- Language and terminology understood by all
  - (Chin, Rosson and Carroll 1997, Karat 1995)
- Closed / No external references
  - (Kyng 1995)
- Reflect reality and solve the right problem
  - (Nardi 1995; Rumbaugh 1994; Chin, Rosson and Carroll 1997)
- Short
  - (Nardi 1995)
- Fun
  - (Nardi 1995)
- Vivid
  - (Nardi 1995)
- High data quality
  - (Nardi 1995)
- Single Threaded
  - (Rumbaugh 1994)

From Figure 17 it is obvious that not everyone believes in the same traits for scenarios. Some characteristics even conflict (such as ambiguous / concrete and complete / incomplete).

As mentioned above, modeling timing behavior or constraints is a good quality to have of a scenario (Kramer and Keng 1988). Dasarathy has researched methods of expressing and validating timing constraints of real-time systems (although not directly through the use of scenarios). He defines two types of timing constraints: those that are performance related (limiting responses of the system) and behavioral related (limiting users' stimuli). The three types of temporal restrictions that can be placed on timing constraints are: maximum, minimum or durational, leading to a combination of stimuli/response max/min timing constraints. Dasarathy models the timing constraints via Finite State Machines and languages such as RTRL (Real Time Requirements Language – from GTE) and the ATLAS test language. He uses such functions or primitives as 'interrupt', 'latency', 'timer', 'delay' and state transitions to model maximum, minimum and durational constraints for a system. (Dasarathy 1985)

Attributes Of a Quality Software Requirements Specification (SRS)

The Software Requirements Specification (SRS) captures the results of requirements elicitation from using such tools as scenarios. Research has been done into what makes a quality SRS. Looking at the attributes of a quality SRS may give some insight into the attributes of a quality scenario since the SRS is the end result of using scenarios. Table 3 shows the different attributes that are believed to make a quality SRS along with the authors that desire those attributes. When available, a definition is also given. These attributes are shown in alphabetical order.

Table 3 Attributes of a Quality SRS

ATTRIBUTE	AUTHOR(s)
Achievable (there could exist at least one system design and implementation that correctly implements all the requirements stated in the SRS (A. Davis and Overmyer et al 1993))	A. Davis and Overmyer et al. 1993
Adaptable to changes in the nature of the needs being satisfied by the component	Roman 1985
Annotated by relative importance (reader can easily determine which requirements are of most importance to customers, which are next important, etc. (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Annotated by relative stability (reader can easily determine which requirements are of most likely to change, which are next most likely, etc. (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Annotated by version (reader can easily determine which requirements will be satisfied by which versions of the product (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Appropriateness (SRS captures, in a manner that is straightforward and free of implementation considerations, those concepts that are germane to the system's role in the environment for which it is intended (Roman 1985)	Roman 1985

ATTRIBUTE	AUTHOR(s)
At right level of Abstraction / Detail	A. Davis and Overmyer et al. 1993
Complete (exhaust all known needs and objectives) Farbey 1990)) (everything the software is supposed to do is included in the SRS; responses of the software to all realizable classes of input data in all realizable classes of situations included; all pages, figures and tables numbered, named and referenced; all terms defined; all units of measure provided and all reference material present; no pages marked 'To Be Determined' (A. Davis and Overmyer et al. 1993))	Holbrook 1990, Farbey 1990, A. Davis and Overmyer et al. 1993, Roman 1985
Concise / Economy of expression (SRS is as short as possible without adversely affecting any other quality of the SRS (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993, Roman 1985
Consistent (*Internally consistent*: no subset of individual requirements stated therein conflict; *Externally consistent*: no requirements stated therein conflicts with any already baselined project documentation (A. Davis and Overmyer et al. 1993))	Holbrook 1990, Farbey 1990, A. Davis and Overmyer et al. 1993, Roman 1985
Constructability (there exists a systematic approach to formulating the requirements (potentially computer assisted) (Roman 1985))	Roman 1985
Correct (absence of incompleteness and redundancy (Davis and Rauscher 1979)) (every requirement represents something required of the system to be built (A. Davis and Overmyer et al. 1993))	Davis and Rauscher 1979, A. Davis and Overmyer et al. 1993, Farbey 1990

ATTRIBUTE	**AUTHOR(s)**
Cross-referenced (cross-references are used in the SRS to relate sections containing requirements to other sections containing: identical / redundant requirements; more abstract or more detailed descriptions of the same requirements; requirements that depend on them or on which they depend (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Design Independent (there exist more than one system design and implementation that correctly implements all requirements in the SRS (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Effective (does the SRS solve the right problem? (Farbey 1990))	Farbey 1990
Electronically stored (the entire SRS is in a word processor, it has been generated from a requirements database or has been otherwise synthesized from some other form (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Executable/interpretable/ prototypable (there exists a software tool capable of inputting the SRS and providing a dynamic behavioral model (A. Davis and Overmyer et al. 1993)) (functional simulations can be constructed from its requirements specification prior to starting the design or implementation (Roman 1985))	A. Davis and Overmyer et al. 1993, Roman 1985
Modifiable / Maintainable (structure and style is such that any changes can be made easily, completely	Holbrook 1990, Farbey 1990, A. Davis and Overmyer et al. 1993, Roman 1985

ATTRIBUTE	AUTHOR(s)
and consistently (A. Davis and Overmyer et al. 1993))	
Not Redundant (SRS is redundant if the same requirement is stated more than once (A. Davis and Overmyer et al. 1993))	A. Davis Overmyer et al. 1993
Organized (contents are arranged so that readers can easily locate information and logical relationships among adjacent sections is apparent (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Performance constraints captured	Farbey 1990, Roman 1985
Precise (numeric quantities are used whenever possible and the appropriate levels of precision are used for all numeric quantities (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993, Roman 1985
Predictable / Testable / Verifiable (cost-effective procedures exist that allow one to verify if the design and/or realization of some component satisfies its functional and non-functional requirements (Roman 1990)) (there exists finite, cost-effective techniques that can be used to verify that every requirement stated therein is satisfied as built (A. Davis and Overmyer et al. 1993))	Farbey 1990, Roman 1985, Holbrook 1990, A. Davis and Overmyer et al. 1993
Readable	Farbey 1990
Reusable (sentences, paragraphs and sections can be easily adopted or adapted for use in a subsequent SRS (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Serviceable	Farbey 1990

ATTRIBUTE	AUTHOR(s)
(provides a firm basis from which to proceed (Farbey 1990))	
Tolerant of temporary incompleteness	Roman 1985
Traceable (SRS is written in a manner that facilitates the referencing of each individual requirement (A. Davis and Overmyer et al. 1993)) (ability to cross-reference items in the requirements specification with items in the design specification (Roman 1985))	Holbrook 1990, Farbey 1990, A. Davis and Overmyer et al. 1993, Roman 1985
Traced (origin of each of the requirements is clear (A. Davis and Overmyer et al. 1993))	A. Davis and Overmyer et al. 1993
Unambiguous (every requirement stated therein has only one possible interpretation (A. Davis and Overmyer et al. 1993)) (two or more interpretations cannot be attached to a particular requirement (Roman 1985))	Holbrook 1990, Farbey 1990, A. Davis and Overmyer et al. 1993, Roman 1985
Useable after implementation	Farbey 1990
Useable	Holbrook 1990

Table 3 shows 32 attributes that different authors believe are important for a quality SRS. Some definitions are similar (such as effective / appropriate and electronically stored / constructable). They all strive to create a quality SRS; one that contributes to successful, cost-effective creation of software that solves real user needs (A. Davis and Overmyer et al. 1993). It would be a near miracle to create an SRS that contained all these attributes because in order to achieve one attribute, another attribute may suffer (such as unambiguous vs. concise). Instead, trade-offs must exist among the attributes.

CHAPTER 3

APPROACH AND RESULTS

Approach

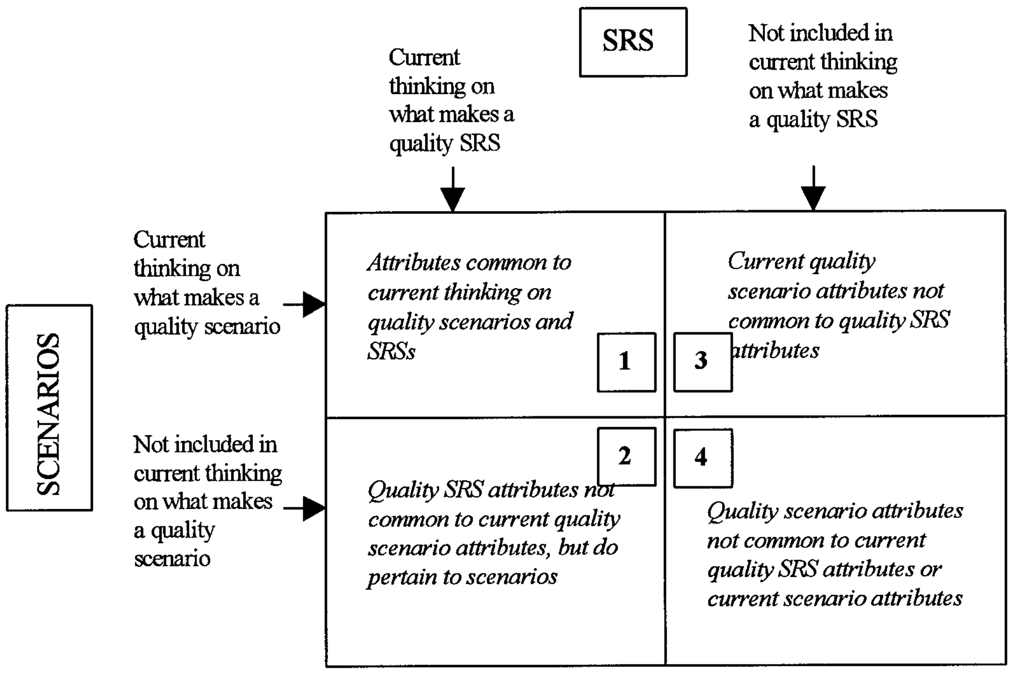
This thesis defines and justifies attributes for a quality scenario and scenario set. There are some attributes that are applicable to a set of scenarios, and not to a single scenario (such as the attribute 'scenario set consistency' defined on page 56). Both single scenario and scenario set attributes will be defined.

Figure 18 shows the sources used to create the attribute list in this thesis. Quadrant 1 represents those attributes that are common to current thinking of what makes a quality scenario and a quality SRS. Quadrant 2 represents quality attributes of an SRS that currently are not attributes of scenarios. Quadrant 3 represents current scenario attributes that are not common to SRS attributes. Quadrant 4 represents attributes that are not included in the current thinking of quality scenarios or quality SRS and are new or fresh ideas on scenario quality. To create the attribute list, I will look first to sources in quadrant 1, then quadrant 2, quadrant 3, and then new thinking in quadrant 4. For the purpose of scope and focus, the scenarios in this paper will be represented in written form.

My approach to my thesis is to list, define and justify attributes of a quality scenario/scenario set. To achieve this, I will give a definition of a scenario and define the goal of scenarios used in requirements elicitation. I will also discuss the idea or symbology behind the 'requirements funnel'. I will describe this funnel since where one is located within the funnel may affect the attributes one wants for their scenario.

After scenario, goal and funnel definitions, I will synthesize previous work done in the area of scenarios and in the area of quality attributes of an SRS (as mentioned in chapter 2). I will examine and expand upon those quality attributes that are common to both a good scenario and a quality SRS (quadrant 1 in Figure 18) for consideration as candidates for the attribute list of what makes a quality scenario/scenario set. Then I will look at attributes that are not common to both (quadrants 2 and 3 in Figure 18) and expand upon and add the appropriate attributes to my list. Next, I will create and add to the list 'new' attributes, or attributes that are fitting for a quality scenario/scenario set, but have not been mentioned before for scenarios or SRSs (quadrant 4 in Figure 18).

Figure 18 Thesis Sources: Current Thinking on Quality Attributes of Scenarios (Quadrants 1 and 3), Quality Attributes of an SRS (Quadrants 1 and 2) and New Thinking (Quadrant 4)


SRS
Current thinking on what makes a quality SRS
Not included in current thinking on what makes a quality SRS
SCENARIOS
Current thinking on what makes a quality scenario
Not included in current thinking on what makes a quality scenario
Attributes common to current thinking on quality scenarios and SRSs
1
Current quality scenario attributes not common to quality SRS attributes
3
2
Quality SRS attributes not common to current quality scenario attributes, but do pertain to scenarios
4
Quality scenario attributes not common to current quality SRS attributes or current scenario attributes

All of the above will be accomplished in this chapter. The subsequent chapter, chapter 4, will validate the attribute list defined in this chapter. Chapter 4 will explain how each attribute is necessary. Necessity will be accomplished by showing if an attribute is not included in a scenario or set of scenarios then the software lifecycle, or other software functions, will be adversely affected.

Results

Definition of a Scenario

As Table 1 in chapter 2 points out, there are many definitions for a scenario. The definition that seems most accurate without dictating a methodology is 'a sequence of interactions between a solution system and its environment which serves as a representative example of how some system feature works' (UCCS 1997). This is the definition that will be used throughout this thesis.

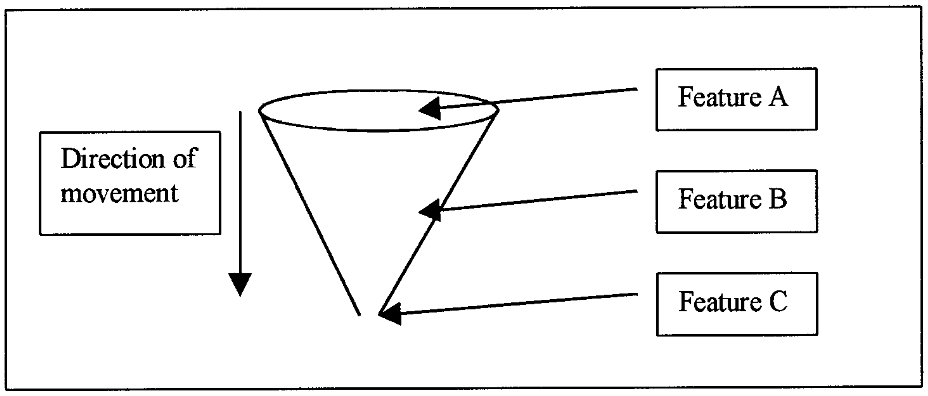
Goal of Using Scenarios for Requirements Elicitation

The goal for using scenarios for the purpose of requirements elicitation is: to gain as much knowledge as possible of what the stakeholders want the proposed system to do and how they want it to behave. Scenarios do this by allowing designers and stakeholders to communicate their 'view' of the proposed system through sample interactions with the future system, correcting and adapting the scenarios as requirements are surfaced and refined. Scenarios help bring out tacit or implied knowledge, a type of knowledge that is very hard for other requirements elicitation techniques to capture. They also help bring into the open any assumptions that the stakeholders may have by problematizing the situation (see page 16 for a definition of problematizing).

The Requirements Funnel

The requirements elicitation process can be thought of as a funnel as shown in Figure 19. The top, wide portion of the funnel represents the beginning steps of finding out the requirements of the system. In the beginning, there are many options or possible ways to build the system until the requirements team learns exactly what the users want. As they learn more and more, the options become fewer and fewer and the understanding of the system becomes more and more refined. This is symbolized by moving down the funnel as it narrows, showing the reduction of options and refinement of requirements. The requirements team may learn about different features of the system at different times. While they may be refining the requirements of some features (such as feature C in Figure 19), they may be still at the top of the funnel with others (such as feature A in Figure 19).

Figure 19 The Requirements Funnel



Where the requirements team is located in the funnel may affect the type of attributes desired in their scenarios. For example, as a requirements engineer, if you are just starting out on the project (at the top of the funnel) and have little domain knowledge you may want scenarios that are a little rough or ambiguous. Two reasons for this are: 1) you do not know enough information to create a concrete scenario and 2) rough or ambiguous scenarios have gaps in them that can be filled in by the customers. As the customers fill in the missing gaps, you gain more insight and information into the domain and requirements of the future system, allowing you to eliminate some options and move further down the funnel. If you tried to 'guess' and create a specific, concrete scenario you may miss the boat completely and waste time spinning your wheels on something non-applicable.

After gaining requirements information from users via rough scenarios, the requirements team moves 'down' the funnel and the scenarios can be refined and made more concrete, allowing for detailed and specific scenarios to be created. The benefit of concrete scenarios is that users work in the concrete or specific and not the abstract (Kyng 1995). The more specific the scenario, the more refined the requirements will be.

In the following sections, each attribute listed will give a 'tolerance for exclusion' rating (low, medium or high). Attributes with a high tolerance for exclusion mean there is quite a bit of leniency in not including this attribute at the beginning of the requirements funnel, or not to include the attribute to its fullest degree at the funnel top. Attributes with a low tolerance for exclusion mean the attribute should be included from the beginning or top of the funnel.

How to Describe the Attributes

The rest of this chapter explains the attributes of a quality scenario / scenario set. Some of the attributes are applicable to a set of scenarios; some are applicable to a single scenario. As mentioned in chapter 2, Table 2, there are many possible representations for scenarios. For the purpose of this thesis, scenarios will be represented in a written form.

Each attribute description includes several subsections:

1 The definition of the attribute is given.

2 The origin of the attribute explains if the attribute is new, or an idea spawned off from a similar definition of an SRS or scenario attribute as described in chapter 2.

3 Further explanation of the attribute is given as needed.

4 The tolerance for exclusion is a rating of how tolerant a quality scenario can be to not include this attribute, or to not include it completely (see previous paragraph for a further explanation).

5 The justification for inclusion describes why the attribute is needed.

6 Each attribute description also provides two examples: one example is what a scenario or set of scenarios would be like if the attribute was not included, and the other example shows the improved scenario or set of scenarios that contains the attribute. The examples may highlight just a *segment* of a scenario to show the inclusion or exclusion of the attribute and may or may not include other attributes.

7 Lastly, each attribute is marked whether the attribute applies to a single scenario or a set of scenarios.

Attributes Common to a Quality SRS and Current Thinking of What Makes a Good Scenario

Table 4 combines the information in Figure 17 Attributes of a Good Scenario and Table 3 Attributes of a Quality SRS in chapter 2. The scenario attributes are column headings (horizontal) and the SRS attributes are row headings (vertical). The common attributes are marked with a check and include: complete, concise / short, consistent, correct, capture performance or timing constraints, right level of abstraction / detail (ambiguous or concrete), understandable / readable and tolerant of temporary incompleteness / incomplete.

Table 4 Current Thinking on Good Scenario Attributes and Attributes of a Quality SRS

Scenario → SRS ↓	timing behavior modeled	correct	consistent	complete	incomplete	validated	concrete / specific	ambiguous / rough in beginning	closed / no external references	reflect reality and solve the right problem	Understand-able	short	fun	vivid	Initial Conditions	Sngl Thread	Maintain data quality
achievable																	
adaptable to changes in the nature of the needs being satisfied																	
annotated by relative importance																	
annotated by relative stability																	
annotated by version																	
appropriate-ness																	
at right level of abstraction / detail							✓	✓									
complete				✓													
concise / economy of expression												✓					
consistent			✓														

Scenario → SRS ↓	timing behav-ior model-ed	correct	consistent	complete	incomplete	validated	concrete / specific	ambig-uous / rough in beginning	closed / no external references	reflect reality and solve the right problem	Understand -able	short	fun	vivid	Initial Condi tions	Sngl Thre ad	Main tain data quali ty
construct-ability																	
correct		✓															
cross-referenced																	
design indepen dent																	
effective																	
electronic-ally stored																	
executable/ interpret-able/ prototyp-able																	
modifiable/ maintain-able																	
not redundant																	
organized																	
perform-ance constraints captured	✓																
precise																	
predictable/ testable/ verifiable																	

Table 4 (Continued)

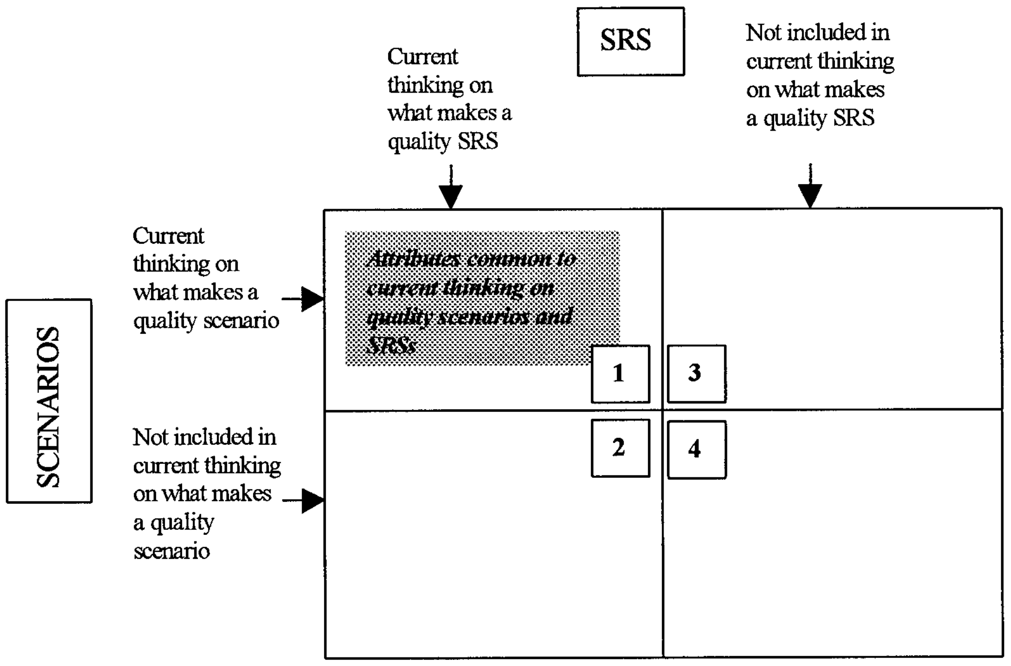
Scenario → SRS ↓	timing behavior modeled	correct	consistent	complete	incomplete	validated	concrete / specific	ambiguous / rough in beginning	closed / no external references	reflect reality and solve the right problem	Understand-able	short	fun	vivid	Initial Conditions	Sngl Thread	Maintain data quality
readable											✓						
reusable																	
serviceable																	
tolerant of temporary incompleteness					✓												
traceable																	
traced																	
unambiguous																	
usable after implementation																	
usable																	

Table 4 (Continued)

The attributes listed on page 43 represent the intersection of current thinking on good scenario traits and attributes of a quality SRS as stated in chapter 2 and represent quadrant 1 in Figure 20. From the intersection attributes, I propose the attributes for a quality scenario/scenario set to include: complete scenario, complete scenario set, concise, discrete, single scenario consistency, scenario set consistency, timing constraints modeled, right level of abstraction / detail, and understandable.

The attribute 'correct' is not included in this set of attributes for a quality scenario/scenario set although it is in quadrant 1 of Figure 20. In order for a scenario to be correct, users and designers need to validate that the scenario is correct. 'Validated' is an attribute that takes the place of 'correct' and is described on page 79.

Figure 20 Attributes Common to Current Thinking on Quality Scenarios and SRSs


SRS
Current thinking on what makes a quality SRS
Not included in current thinking on what makes a quality SRS
SCENARIOS
Current thinking on what makes a quality scenario
Not included in current thinking on what makes a quality scenario
Attributes common to current thinking on quality scenarios and SRSs
1
2
3
4

1. Complete Scenario

- ❖ *1.1 attribute definition:* A complete scenario has: 1) at least one input, 2) at least one output, 3) an external actor or entity performing an action and 4) the future system performing an action.
- ❖ *1.2 origin*: scenario attribute 'complete' (by UCCS 1997).
- ❖ *1.3 further explanation*: A scenario at a minimum needs an input, an output, an external entity performing an action and the system performing an action (most likely the system's response to an external input) that accomplishes a specific goal.
- ❖ *1.4 tolerance for exclusion*: MEDIUM. An incomplete scenario is not very meaningful, however, a scenario needs to be tolerant of temporary incompleteness especially at the top of the requirements funnel. This allows scenario designers to model what they know while leaving unknown areas marked 'TBD' to be filled in by the team at a later date. A finalized scenario should be complete.
- ❖ *1.5 justification for inclusion*: Without inputs, it is unknown what is causing the scenario to occur. Without outputs, it is unclear what the scenario is accomplishing. If the system you are modeling is not in the scenario, then the scenario is of no interest for requirements elicitation. An external entity (human, sensor, other computer, etc.) must provide the input to the system.

❖ *1.6 Example* (domain- bank loan program):

Attribute not present	Attribute present
EXAMPLE: Initial Conditions [page 77]: System is in 'Calculate Loan Payment' mode • Customer enters principal amount and number of years for loan • System calculates monthly payment	EXAMPLE: Initial Conditions [page 77]: System is in 'Calculate Loan Payment' mode • Customer enters principal amount and number of years for loan • System calculates monthly payment • System prints out payment schedule
EXPLANATION: This scenario is not complete because there is no output (just the internal action of calculating monthly payment that the system takes)	EXPLANATION: This scenario is complete because the customer is the external entity, the input is the principal and # of years, the system performs the action of calculating monthly payment and outputting the payment schedule.

❖ *1.7 applicable to*: ☐ set of scenarios [✓ single scenario]

2. Complete Scenario Set

❖ *2.1 attribute definition:* For each possible state the system can be in, the scenario set shows what happens when each possible input is received.

- *2.2 origin*: SRS attribute definition of 'complete' (by Farbey 1990): 'to exhaust all known needs and objectives' and scenario attribute 'complete' (by UCCS 1997).
- *2.3 further explanation*: All actors' (both human and computer) interactions with the future system are modeled as appropriate (see the definition of appropriate on page 67).
- *2.4 tolerance for exclusion*: HIGH. Scenario sets need to be tolerant of temporary incompleteness especially at the top of the requirements funnel. This allows scenario designers to model what they know while leaving unknown areas marked 'TBD' to be filled in by the team at a later date. A set of scenarios will not need to be complete until the bottom of the requirements funnel.
- *2.5 justification for inclusion*: It is important to have a complete set of scenarios to make sure requirements are covered, to help eliminate assumptions made about missing information, and to bring tacit knowledge into the open.
- *2.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: A set of scenarios for an ATM machine models users desiring a withdrawal and depositing money, but does not model what happens when a user wants to check their balance.	EXAMPLE: A set of scenarios for an ATM machine models all possible user interactions with the system including a user checking his/her balance.

- *2.7 applicable to*: [✓ ☒] set of scenarios ☐ single scenario

3. Concise

- ❖ *3.1 attribute definition:* KISS: Keep It Short and Simple
- ❖ *3.2 origin*: SRS attribute ‘concise’ (by A. Davis and Overmyer et al. 1993): ‘SRS is short as possible without affecting any other quality of the SRS’; and scenario attribute ‘short’ (by Nardi 1995).
- ❖ *3.3 further explanation*: Scenario steps that are long and complex may be difficult to follow or understand exactly what the step is accomplishing. Short, simple scenarios that are to the point get the message of the scenario across to the reader succinctly.
- ❖ *3.4 tolerance for exclusion*: LOW. Where one is located in the requirements funnel does not have an effect on keeping the scenario concise. Scenarios should be concise from the beginning.
- ❖ *3.5 justification for inclusion*: Scenarios are very user oriented. It is important to keep the stakeholders involved and not bored or buried in too much detail so that the stakeholders maintain interest.

❖ *3.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: • Prompting the user, the system requests the user insert their ATM card into the ATM card reader • User inserts ATM card that was issued by customer's bank and includes magnetic strip encoding PIN and card number and a card number in raised letters on top of card • Prompting the user, the system requests the user enter their PIN • User uses keyboard located below the monitor to enter 4 digit Personal Identification Number • After receiving the last digit, the system performs PIN verification • Main menu of options is displayed on the screen by the system • 'withdrawal' button is pressed by the user • Prompting the user, the system requests the user to enter their amount they want to withdraw • User enters amount of money he/she wants to withdraw by using keyboard located below monitor • System verifies funds by sending electronic message to customer's bank's computer where the customer's bank's computer checks that the customer's account has the desired withdrawal amount and sends and electronic message to the ATM machine that the customer has enough money to cover the withdrawal • Dispersing cash, the system outputs the withdrawal amount desired by the user	EXAMPLE: • System prompts user to insert ATM card • User inserts ATM card • System prompts user to enter PIN • System verifies PIN • System displays main menu of options • User presses 'withdrawal' button • System prompts user to enter amount • User enters amount • System verifies funds • System disperses cash

EXPLANATION: This scenario for an ATM program is not concise, the steps are not as short as possible.	EXPLANATION: This scenario captures the system and customer interactions concisely.

❖ *3.7 applicable to*: ☐ set of scenarios ☒ single scenario

4. Discrete

❖ *4.1 attribute definition:* Scenario accomplishes a single goal or transaction.

❖ *4.2 origin*: SRS attribute 'concise' (by A. Davis and Overmyer et al. 1993): 'SRS is short as possible without affecting any other quality of the SRS'; and scenario attribute 'short' (by Nardi 1995).

❖ *4.3 further explanation*: At times, users of a system may accomplish 2 or more goals or transactions consecutively when interacting with a system. Each of these individual goals or transactions should have its own scenario.

❖ *4.4 tolerance for exclusion*: LOW. Where one is located in the requirements funnel does not have an effect on keeping the scenario discrete. Scenarios should be discrete from the beginning.

❖ *4.5 justification for inclusion*: Scenarios that accomplish more than one goal or transaction may become long and unmanageable and may result in 'losing the user' from too much information or boredom.

❖ 4.6 *Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: • System prompts user to insert ATM card • User inserts ATM card • System prompts user to enter PIN • System verifies PIN • System displays main menu of options • User presses 'withdrawal' button • System prompts user to enter amount • User enters amount • System verifies funds • System disperses cash • System prompts user asking if they want another transaction • User presses 'yes' button • System displays main menu of options • User presses 'deposit' button • System prompts user to enter amount • User enters amount • System prompts user to enter deposit through deposit slot • User inserts deposit • System prompts user asking if they want another transaction • User presses 'no' button	EXAMPLE: Name: Normal cash withdrawal with second transaction desired • System prompts user to insert ATM card • User inserts ATM card • System prompts user to enter PIN • System verifies PIN • System displays main menu of options • User presses 'withdrawal' button • System prompts user to enter amount • User enters amount • System verifies funds • System disperses cash • System prompts user asking if they want another transaction • User presses 'yes' button Name: transaction completed, user desires to make a deposit Initial Conditions[page 77]: User has successfully completed a transaction and user has pressed 'yes' button for another transaction • System displays main menu of options • User presses 'deposit' button • System prompts user to enter amount • User enters amount • System prompts user to enter deposit through deposit slot • User inserts deposit • System prompts user asking if they want another transaction • User presses 'no' button

EXPLANATION: This scenario for an ATM program is not discrete and can be logically broken into two separate scenarios (one for the withdrawal and one for the deposit) .	EXPLANATION: The long scenario is broken into 2 logical scenarios. Each scenario is discrete and accomplishes a single goal. The first scenario shows how to withdraw cash and still desire second transaction and the second scenario shows how to deposit money after already completing a transaction.

- *4.7 applicable to*: [ ] set of scenarios [✓ single scenario]

5. Single Scenario Consistency

- *5.1 attribute definition:* The scenario does not have any contradictions within the scenario itself.
- *5.2 origin*: SRS attribute 'internally consistent' (by A. Davis and Overmyer et al. 1993): 'no subset of individual requirements stated therein conflict' and scenario attribute 'consistent' (by Hsia et al. 1994 and UCCS 1997).
- *5.3 further explanation*: Within the scenario itself there are no contradictions or inconsistencies such as in the example below. Is the alert button a push button or something to be turned?
- *5.4 tolerance for exclusion*: LOW. There is no reason why a scenario should be inconsistent from the beginning.
- *5.5 justification for inclusion*: Scenarios that are inconsistent can lead to multiple interpretations that may lead to a system built that satisfies inconsistent requirements.

- *5.6 Example* (domain – Alert System)*:*

Attribute not present	Attribute present
EXAMPLE: • User presses the alert button (to 'on' state) • The system's status bar turns red • User turns the alert button off	EXAMPLE: • User presses the alert button (to 'on' state) • The system's status bar turns red • User presses the alert button (to 'off' state)
EXPLANATION: This scenario segment is inconsistent because the first bullet entails the alert button is pressed and the last bullet describes the alert button as being a turn button.	EXPLANATION: This scenario segment is not inconsistent because both times the alert button is mentioned, it is pressed.

- *5.7 applicable to*: ☐ set of scenarios ☒ single scenario

6. Scenario Set Consistency

- *6.1 attribute definition:* There is no contradiction among the set of scenarios.
- *6.2 origin*: SRS attribute 'internally consistent' (by A. Davis and Overmyer et al. 1993): 'no subset of individual requirements stated therein conflict' and scenario attribute 'consistent' (by Hsia et al. 1994 and UCCS 1997).
- *6.3 further explanation*: This attribute implies that the same sequence of steps or inputs (assuming identical initial conditions - see page 77 for definition of initial conditions) cannot generate two different, contradictory outputs in 2 or more scenarios.

- *6.4 tolerance for exclusion*: MEDIUM. There may be times when a designer wants to show the user different output alternatives for a sequence of inputs and therefore may have inconsistent scenarios. However, a finalized set of scenarios (with the desired alternative already chosen) should not have inconsistent scenarios.
- *6.5 justification for inclusion*: One does not want inconsistent requirements. They may lead to an ambiguous or non-verifiable SRS.
- *6.6 Example* (domain – Missile Launch System):

This is an example of 2 inconsistent scenarios. This example shows two different scenarios with the same initial conditions and inputs, but drastically different outputs. The scenario on the right is probably the more correct scenario	
EXAMPLE: Initial Conditions[page 77]: System is in launch mode • Missile Operator presses launch button • Missile Operator presses cancel button within 30 seconds of pressing red button • Missile is launched	EXAMPLE: Initial Conditions[page 77]: System is in launch mode • Missile Operator presses launch button • Missile Operator presses cancel button within 30 seconds of pressing red button • Launch is aborted • System returns to normal state
EXPLANATION: The inputs to this scenario are: user presses red button, user presses cancel button within 30 seconds of pressing red button. The output is: the missile is launched.	EXPLANATION: The inputs to this scenario are: user presses red button, user presses cancel button within 30 seconds of pressing red button. The output is: the launch is aborted.

- *6.7 applicable to*: [✓ set of scenarios] ☐ single scenario

7. Timing Constraints Modeled

- ❖ *7.1 attribute definition:* Scenarios model timing constraints between stimuli and responses, showing maximum or minimum time from when a stimulus or response is received or sent by the system to when another response or stimulus occurs.
- ❖ *7.2 origin*: scenario attribute 'timing behavior modeled' (by Kramer and Keng 1988 and Neilsen 1995) and SRS attribute 'performance constraints captured' (by Farbey 1990 and Roman 1985).
- ❖ *7.3 further explanation*: Users should be able to get a feel for the timing constraints via the scenario and be able to change the values of the constraints. There should exist the ability to turn on or off the timing constraint modeling capability.
- ❖ *7.4 tolerance for exclusion*: LOW to HIGH. LOW: For real time systems, the timing is crucial and is one of the first features of the system to be described. HIGH: For other systems, most time and energy at the beginning of elicitation is spent on capturing ideas and concepts. Timing behavior is normally not included until the ideas are refined (towards the narrow end of the funnel) although they can be included at any point.
- ❖ *7.5 justification for inclusion*: One of the hardest and often most overlooked areas of requirements are non-functional requirements (including timing constraints). By explicitly modeling timing constraints in scenarios they are more likely to be incorporated into the SRS. By letting the users get a feel for what the constraints are (e.g. how 'painful' is it to wait 5 minutes for an elevator?), the more likely

they are to find the right constraints the users can live with. In addition, discussions of tradeoffs among different constraints can be prompted (i.e. tighter constraints may mean additional hardware and more money, or tightening one constraint means loosening another, etc.). Also, being able to turn off or on the constraint modeling ability allows for the scenario to be presented with the constraints shown (which may make the presentation longer) or, once an initial presentation has been given, to be presented without modeling the constraints. This can help speed up the process when one is looking at a scenario and is focusing on some aspect other than the constraints.

❖ *7.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: • User enters amount to withdraw • System disperses cash requested	EXAMPLE: • User enters amount to withdraw • System disperses cash requested (within 10 seconds of amount to be withdrawn entered by user)
EXPLANATION: This portion of an ATM scenario does not include timing constraints. The user may have to wait until the next day to receive his/her funds.	EXPLANATION: This portion of an ATM scenario does include timing constraints. It is clear that the user should not have to wait more than 10 seconds to receive his/her cash.

❖ *7.7 applicable to*: ☐ set of scenarios ☒ single scenario

8. Right Level of Abstraction / Detail

- ❖ *8.1 attribute definition:* A scenario is detailed enough to be meaningful and demonstrate exceptional cases, yet is abstract enough to avoid generating too many scenarios.
- ❖ *8.2 origin*: SRS attribute 'at right level of abstraction/detail' (by A. Davis and Overmyer et al. 1993) and scenario attributes 'ambiguous' and 'concrete' (by Erickson 1995; Potts, Takahashi, Anton 1994; Carroll 1995; Kyng 1995; Rosson and Carroll 1995).
- ❖ *8.3 further explanation*: There is no consensus on the right level of detail for a scenario (Rosson and Carroll 1995), it is difficult to know when enough detail has been reached. The right level of abstraction or detail is not only very subjective, but also depends on where in the requirements funnel you are. In the beginning or top of the funnel, it is okay and often desirable to have more abstract or ambiguous scenarios because during this time you are trying to get a higher level understanding of what the users want in their system and do not have a lot of domain knowledge. Rough or ambiguous scenarios leave gaps that open up discussion and allow users to tell the designers what they want to happen to fill those gaps. After gaining knowledge into the overall system, it is important to make the scenarios more concrete or detailed in order to flush out the requirements and avoid assumptions. The right level of detail is very domain dependent. Relying on domain expertise to guide the level of detail is important. For example, scenarios could be at the high level 'class of input' level or at the very detailed level of having a different scenario for every specific input possible.

The latter could yield an enormous amount of scenarios (just think of a calculator program as an example).

- *8.4 tolerance for exclusion*: MEDIUM. It may be difficult at first to find the right level of detail. But, a few tries at creating scenarios along with domain expert guidance should soon guide the proper level of detail.
- *8.5 justification for inclusion*: If the scenarios are too detailed then it is easy to get lost in the overflow of information, yet if they are too abstract then the scenarios may not be meaningful enough. If one enumerates every specific possible input as a scenario, then there may be too many scenarios to deal with, yet if too few are enumerated then not enough exceptional cases may be captured, leaving the scenarios incomplete and ambiguous.

❖ *8.6 Example* (domain – network messaging system)

Right level of abstraction for upper requirements funnel (not appropriate for lower funnel)	Right level of detail for lower requirements funnel (may not be appropriate for upper funnel)
EXAMPLE: • User gains access to the system • User enters command to stop outgoing messages • System stops sending outgoing messages	EXAMPLE: • System displays login prompt • User types in login name and password correctly • System displays main menu with options: start outgoing update messages, view outgoing update messages and stop outgoing update messages • User selects 'stop outgoing update messages' • System prompts with an OK button 'are you sure?' • User touches OK • System stops sending outgoing update messages
EXPLANATION: This scenario does not explain how the user gains access to the system, or how he/she stops outgoing messages. This scenario only captures the higher level sequence of actions for stopping outgoing messages (which is all that may be known at the upper portion of the requirements funnel)	EXPLANATION: This scenario explains how the user gains access to the system and how to stop outgoing messages. It is less ambiguous than the previous example.

❖ *8.7 applicable to*: ☐ set of scenarios ☒ single scenario

9. Understandable

- *9.1 attribute definition:* The scenario should be easy to comprehend by all stakeholders involved in the requirements elicitation process.
- *9.2 origin*: SRS attribute 'readable' (by Farbey 1990) and scenario attribute 'understandable' by (Chin, Rosson and Carroll 1997 and Karat 1995).
- *9.3 further explanation*: As mentioned in chapter 2, stakeholders are all of those who have a stake in the change being considered, those who stand to gain and stand to lose (Macaulay 1993). While it is probably not feasible for all possible stakeholders to be involved in the requirements elicitation process, the scenarios should be understandable by all the stakeholders that are involved. This means that if all the stakeholders are engineers and scientists familiar with diagrams and grammars, a formal scenario method, understandable by all participants, may be appropriate. However, if all or most stakeholders are not familiar with formal notation then an informal representation (such as text or storyboarding) may be appropriate. Nearly everyone can understand text or storyboards, but not everyone can understand or are comfortable with a formal representation. Therefore, when in doubt, it is better to go with an informal method.
- *9.4 tolerance for exclusion*: LOW. If stakeholders cannot understand the scenario from the beginning then it will not be very effective.
- *9.5 justification for inclusion*: The idea behind scenarios is to get users involved with the elicitation process. Users need to feel comfortable with understanding and manipulating what they see. If they cannot comprehend the scenario they will be less likely to make changes or agree with what they see since most time will be

spent trying to understand the scenario. In addition, there is an intimidation factor. Stakeholders may feel intimidated by notation they are unfamiliar with (even after the notation has been explained to them). They may feel they are not smart enough to understand or even participate, resulting in lost ideas from that stakeholder. The idea is to make the stakeholders feel as comfortable as possible so that they participate with their ideas.

❖ *9.6 Example* (domain – telephone system)

Attribute not present	Attribute present

EXAMPLE:
 	
G_v = (NT, S, R, A)
 	
S = {Off_H. Not9, d, R, Cpu, talk, Chup, On_H}
 	
NT= {Caller, <FD>, <Int>, <Third>, <Fourth>, <Action>, <Talk>, <Fin>}
 	
A= Caller
 	
R= {Caller -> Off_H <DialTone>, <DialTone> -> Not9<InternalCall>, <Int> -> digit<Third> | On-H <Third> -> digit<Fourth> | On-H, <Fourth> -> digit<Validating> | On-H, <Validating> ->Ring<Connecting> | On_H | Busy<TryAgain> | <disconnected>
 	
(example taken from Hsia et al. 1994)
	EXAMPLE:
	
	Initial Conditions[page 77]:
	Callee is on the phone
	Name: Caller attempts to call someone who is already on the phone
	• Caller takes the phone off the hook
	• Caller hears dial tone from phone
	• Caller enters a digit other than 9 for an internal call
	• Caller enters 3 more digits
	• System verifies phone number is valid
	• Caller is given a busy signal
	• Caller puts phone on the hook

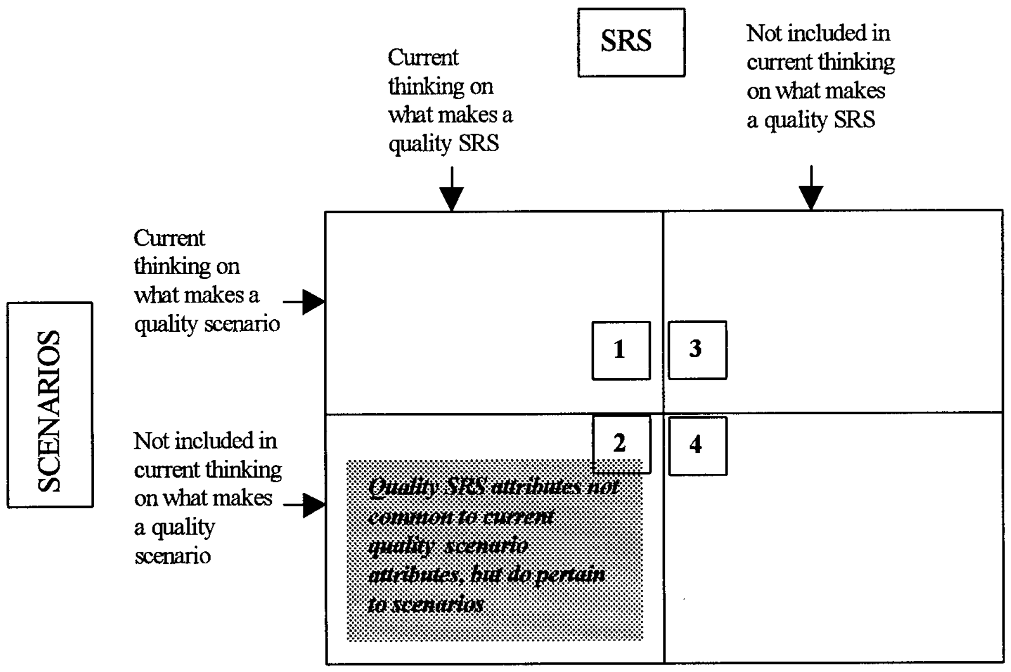
EXPLANATION: For stakeholders that are not familiar with formal grammars, this scenario may be intimidating.	EXPLANATION: For stakeholders that are not familiar with formal grammars, they may be more comfortable with this format of a scenario.

❖ *9.7 applicable to*: set of scenarios [✓ single scenario]

Quality Attributes of an SRS not Previously Mentioned for Scenarios, but do Apply to Scenarios

There are several attributes of a quality SRS that are not common with those mentioned by authors to be important for a scenario (represented by quadrant 2 in Figure 21). Yet, upon examining them, some seem important and applicable to scenarios. Those attributes are: achievable, appropriate, predictable/testable/verifiable, usable after implementation, traceable, and traced. From these attributes, I propose the following attributes for a quality scenario:

Figure 21 SRS Attributes not Common with Current Thinking on Scenario Attributes


SRS
Current thinking on what makes a quality SRS
Not included in current thinking on what makes a quality SRS
SCENARIOS
Current thinking on what makes a quality scenario
Not included in current thinking on what makes a quality scenario
1
2
3
4
Quality SRS attributes not common to current quality scenario attributes, but do pertain to scenarios

10. Achievable

- *10.1 attribute definition:* There exists at least one system design and implementation that correctly implements the scenario.
- *10.2 origin*: SRS attribute 'achievable' (by A. Davis and Overmyer et al. 1993): 'there could exist at least one system design and implementation that correctly implements all the requirements stated in the SRS'.
- *10.3 further explanation*: N/A
- *10.4 tolerance for exclusion*: LOW to HIGH. LOW: Designers do not want to build up users' expectations of a feature or a scenario to later tell them the system cannot be built as shown. HIGH (if brainstorming): In the very beginning of requirements elicitation, users may not know what they want from their future system. Brainstorming ideas and scenarios may help users and designers decide on which features they want. However, all brainstormed ideas may not be achievable and this is okay at the very, very top of the requirements funnel until stakeholders decide on the features they want. After a feature is decided on (simulating a move down the funnel), the feature needs to be achievable.
- *10.5 justification for inclusion*: It is important not to build up users' expectations for something that is infeasible to build (for either technology or scope reasons).

❖ *10.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: • System displays prompt for PIN • Customer enters PIN • System verifies PIN • System displays main menu options • Customer presses account balance button • Customer is shown account balance, last 10 transactions on account and minimum balance needed for that account	EXAMPLE: • System prompts user for PIN • Customer enters PIN • System verifies PIN • System displays main menu options • Customer presses account balance button • Customer is shown account balance
EXPLANATION: The project (ATM system) has a limited amount of bandwidth for transferring information over the line. There is not enough bandwidth available to get the customer's last 10 transactions and minimum balance (only current balance).	EXPLANATION: The project (ATM system) has a limited amount of bandwidth for transferring information over the line. There is not enough bandwidth available to get the customer's last 10 transactions and minimum balance (only current balance).

❖ *10.7 applicable to*: ☐ set of scenarios [✓ single scenario]

11. Appropriate

❖ *11.1 attribute definition:* The scenario only captures the concepts, ideas and interactions that are germane to the future system and the environment it will be deployed in.

❖ *11.2 origin*: SRS attribute 'appropriateness' (by Roman 1985): 'SRS captures, in a manner that is straightforward and free of implementation considerations, those concepts that are germane to the system's role in the environment for which it is intended.'

- *11.3 further explanation*: N/A
- *11.4 tolerance for exclusion*: MEDIUM. At the very top of the funnel it may be difficult to know what is appropriate until the scope of the project is further refined. However, keeping focused on the pertinent aspects of the system is important in order to avoid going down the wrong path.
- *11.5 justification for inclusion*: Keeping focused is important, otherwise it is easy to become bogged down in information overflow with parts of scenarios that do not affect the future system requirements or the environment it will be operating in. This attribute helps scope the scenarios.

❖ *11.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
Example: • *Customer drives up in car* • *Customer parks car* • *Customer exits car* • *Customer locks car* • System prompts customer to insert ATM card • Customer inserts ATM card • System displays prompt for PIN • Customer enters PIN • System verifies PIN • System prompts displays main menu of options • Customer presses 'withdrawal' button • System prompts user for amount • Customer enters amount • System verifies funds • System disperses cash • Customer removes cash • System outputs customer's card • Customer removes card • *Customer unlocks car* • *Customer starts car* • *Customer drives away*	Example: • System prompts customer to insert ATM card • Customer inserts ATM card • System displays prompt for PIN • Customer enters PIN • System verifies PIN • System displays main menu of options • Customer presses 'withdrawal' button • System prompts user for amount • Customer enters amount • System verifies funds • System disperses cash • Customer removes cash • System outputs customer's card • Customer removes card
EXPLANATION: When modeling a future ATM system, unneeded information is captured (such as the customer driving up and locking their car, etc. - as shown in *italics*)	EXPLANATION: When modeling a future ATM system, only germane information is captured.

❖ *11.7 applicable to*: ☐ set of scenarios ☒ single scenario

12. Usable after SRS Written {testable / verifiable}

- ❖ *12.1 attribute definition:* If desired, the scenario could be used after the SRS is written as a test case and to drive design.
- ❖ *12.2 origin*: SRS attribute 'useable after implementation' (by Farbey 1990).
- ❖ *12.3 further explanation*: The scenario must be presented/written in such a way that the system, once built, can be tested that it performs as the scenario dictates. In addition, the scenario can be used to drive design (see page 12 for a discussion of use cases in design).
- ❖ *12.4 tolerance for exclusion*: HIGH. Scenarios can accomplish the goal of requirements elicitation without being reusable in later parts of the software lifecycle. This is especially true at the top of the requirements funnel.
- ❖ *12.5 justification for inclusion*: Validated (see page 79 for definition of validated) scenarios are the agreed way the users want the system to behave. Therefore, to ensure the final product behaves as the user expects, it can be tested that it behaves like the validated scenarios.

❖ *12.6 Example* (domain – system using a button to show 'status')

Attribute not present	Attribute present
EXAMPLE: • User presses 'on' button • Button turns red and green	EXAMPLE: Initial Conditions[page 77]: 'on' button is white • User presses 'on' button • Button turns red within one second of being pressed • System warms up • Within 5 seconds of button turning red, button turns green
EXPLANATION: This portion of a scenario does not explain how the changing from red to green occurs. Does it flash red and green? Does it stay red for hours and then turn green at the last second? It is not clear how to test this scenario. In addition, it is unclear how to design the system to accomplish this scenario.	EXPLANATION: This portion of a scenario explains when the button will turn from red to green. What the design of the system should accomplish is clear. A tester can press the 'on' button, test that it turns red within limits and then turns green within limits.

❖ *12.7 applicable to*: ☐ set of scenarios ☒ single scenario

13. Named {Traceable}

❖ *13.1 attribute definition:* Each scenario should be easily identifiable or referenced.

❖ *13.2 origin*: SRS attribute 'traceable' (by A. Davis and Overmyer et al. 1993): 'SRS is written in a manner that facilitates the referencing of each individual requirement.'

- *13.3 further explanation*: N/A
- *13.4 tolerance for exclusion*: LOW. When a scenario is first created it should be named. It does not matter where in the requirements funnel one is.
- *13.5 justification for inclusion*: Some scenarios may be common, or sub-scenarios, to many scenarios, or one scenario may have to occur before another, or one scenario may be an exceptional case to another scenario, etc. In each of these cases there is the need to reference another scenario. Instead of describing the entire scenario again, the scenario's name can be given. The name of the scenario should reflect what the scenario is accomplishing to give readers a feeling for what it is about. In addition, when the SRS is written, it may be desirable to reference which scenario a requirement is satisfying.
- *13.6 Example* (domain – a system with a specific shut down sequence)

Attribute not present	Attribute present
EXAMPLE: • User presses 'off' button • System prompts user with an OK button asking if they are sure they want to shut the system down • User touches OK button • System shuts down	EXAMPLE: NAME: Successful system shut down • User presses 'off' button • System prompts user with an OK button asking if they are sure they want to shut the system down • User touches OK button • System shuts down
EXPLANATION: This scenario does not have a name and cannot be referenced.	EXPLANATION: This scenario is named and can be referenced.

- *13.7 applicable to*: ☐ set of scenarios ☒ single scenario

14. Traced

- *14.1 attribute definition:* Attached to each scenario is any or all of the following information as appropriate: creator, participating stakeholders, date presented, change history, relationships with other scenarios (including scenarios that can be ran in parallel), rationale for scenario, criticality of the scenario, origin from requirements, and issues.
- *14.2 origin*: SRS attribute 'traced' (by A. Davis and Overmyer et al. 1993): 'Origin of the requirements is clear'.
- *14.3 further explanation*: N/A
- *14.4 tolerance for exclusion*: LOW. From the beginning it should be recorded who created the scenario, why it changed, rationale behind the scenario, etc. This does not depend on where in the requirements funnel one is.
- *14.5 justification for inclusion*: One attribute of a quality SRS is for the origin of the requirements to be clear (A. Davis and Overmyer et al. 1993). Since scenarios elicit requirements, it is important to know how the scenarios came to be: who created it, which stakeholders validated it, rationale behind any changes made, etc. Also, to create a scenario, there must exist at least a very high level requirement that the scenario is depicting. That requirement needs to be recorded. When any questions arise about a traced scenario, the answers can be looked up.

❖ *14.6 Example* (domain – a system with a specific start-up sequence)

Attribute not present	Attribute present
EXAMPLE: Name: Successful system start-up • User plugs system in • User presses 'start' button • System displays opening menu within 2 minutes of start button being pressed • User selects 'start system' • Main user menu is displayed and system is operational	EXAMPLE: <u>Background information</u>: Creator: Kim Braun Participating stakeholders: Ann Zweig, Melinda Mello, J.D. Burton Date Presented: Aug 16, 1997 Change History: Aug 1 1997: instead of going to main menu on start up, Ann suggested an opening menu before the main menu with options: start system, perform maintenance, and perform diagnostics Relationship with other scenarios: 'perform diagnostics' and 'perform maintenance' scenarios show other options of what can occur when 'start system' is not selected from opening menu Rationale for scenario: show how system is turned on Criticality: HIGH Scenario Requirement: Requirement 3.4 Other issues: Melinda might like to use a key instead of the start button to start the system Name: Successful system start-up • User plugs system in • User presses 'start' button • System displays opening menu within 2 minutes of start button being pressed • User selects 'start system' • Main user menu is displayed and system is operational

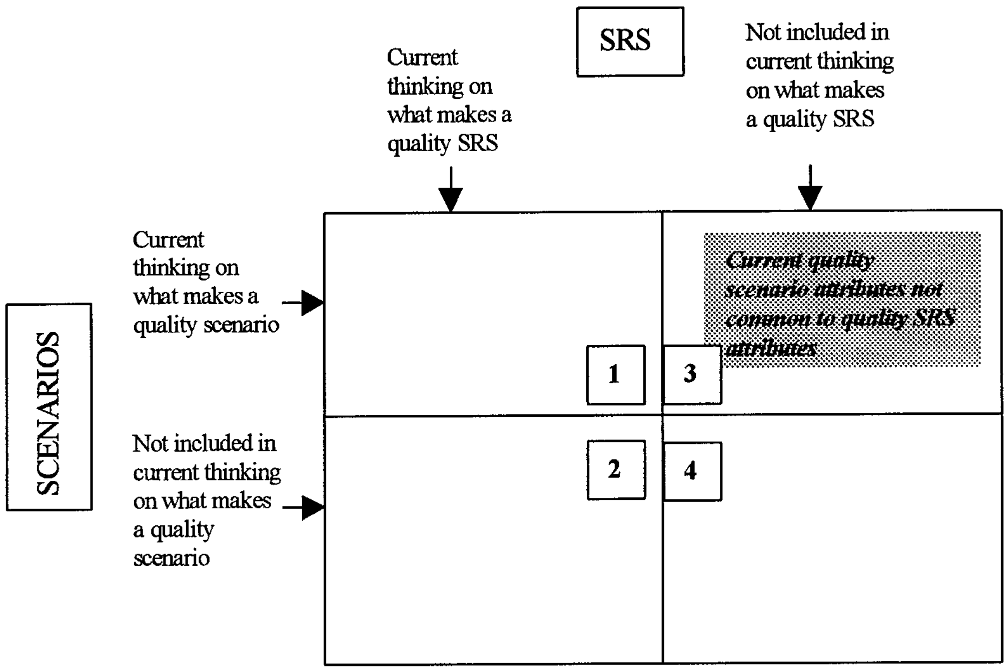
EXPLANATION:	EXPLANATION:
There is no record of information with the scenario.	The scenario has a full record of information attached with it to record its history.

❖ *14.7 applicable to*: ☐ set of scenarios [✓ single scenario]

Current Quality Attributes of Scenarios not Common with Attributes of a Good SRS

There are some attributes that authors feel are important for good scenarios that are not common with attributes of an SRS (represented by quadrant 3 as highlighted in Figure 22). These include: closed/no external references, reflect reality and solve the right problem, fun, vivid, maintain data quality, validated, initial conditions described, and single threaded. I will describe why I believe only three additional attributes should be added to the list from those mentioned above.

Figure 22 Current Scenario Attributes not Common with SRS Attributes


SRS
Current thinking on what makes a quality SRS
Not included in current thinking on what makes a quality SRS
SCENARIOS
Current thinking on what makes a quality scenario
Not included in current thinking on what makes a quality scenario
1
2
3
4
Current quality scenario attributes not common to quality SRS attributes

CLOSED: Kyng's belief is that scenarios should have no external references. However, this may be too restrictive to apply to all scenarios. I prefer the previously mentioned attribute 'appropriate' (page 67) that states scenarios should only model those aspects that are germane to the future system and the environment it will be operating in.

REFLECT REALITY: The attribute 'appropriate' covers the need to reflect reality and solve the right problem. 'Appropriate' reminds us to only model the appropriate or realistic interactions that affect the future system.

FUN: To require all scenarios to be fun may mean they do not reflect reality or are appropriate. When possible, it is a good idea to make scenarios fun since fun scenarios are engaging and hold stakeholders' interests, but it should not be mandated as an attribute.

VIVID: Vivid is a hard term to define in terms of scenarios. It brings to mind such words as detailed, specific, reflect reality, etc. These are terms already covered in the attribute list so far.

MAINTAINING DATA QUALITY: Maintaining data quality is important for scenarios. It means to ensure the data (such as inputs and outputs) are correct. This falls under the attribute 'validated' as described on page 79.

Describing initial conditions sets up the environment and setting that the scenario needs in order to succeed. It is an important attribute to add to the list along with the attributes 'validated' and 'single threaded'.

15. Initial Conditions Described

- *15.1 attribute definition:* A description of what the environment and / or system is like prior to the scenario.
- *15.2 origin*: scenario attribute 'initial conditions' (by UCCS 1997).
- *15.3 further explanation*: An initial condition could be a state that the system is in such as: normal, idle, busy, computing, degraded, etc. (each state would be described at some point). It could be a description of the environment such as 'car has pulled up to the stoplight'. It could even be which scenario had to finish prior to this scenario beginning. Initial conditions could be any combination of the above. If a scenario must have some type of initial conditions met to succeed, they need to be listed.
- *15.4 tolerance for exclusion*: LOW. Initial conditions should be known at time of scenario creation and updated as necessary.

❖ *15.5 justification for inclusion*: It is important to understand that certain conditions must be in place for a scenario to succeed so that stakeholders get a feel for how the scenario fits into the big picture. Also, for testing, testers need to understand that a scenario may fail (and rightly so) if initial conditions are not met.

❖ *15.6 Example* (domain – missile launch system)

Attribute not present	Attribute present
EXAMPLE: Name: Missile Launch • Missile operator turns launch keys • Missile operator presses launch button • Missile is launched	EXAMPLE: Name: Missile Launch Initial Conditions: • System is in launch mode • The 'operators successfully enter launch code ' scenario complete • Launch hatch open Scenario: • Missile operator turns launch keys • Missile operator presses launch button • Missile is launched
EXPLANATION: No initial conditions. This scenario implies that all that is needed to launch a missile is to turn the keys and press the button at any time and under any conditions.	EXPLANATION: With initial conditions describing state of the system, which scenario had to complete before this scenario, and a description of the environment

❖ *15.7 applicable to*: ☐ set of scenarios ☒ single scenario

16. Validated

- ❖ *16.1 attribute definition:* Stakeholders verify the scenario correctly models their beliefs of how the future system should behave and that the data used in the scenario is correct. In addition, designers verify that it meets their quality attribute goals (i.e. is the scenario feasible to implement?, does it contradict another scenario?).
- ❖ *16.2 origin*: scenario attribute 'validated' (by Hsia et al. 1994).
- ❖ *16.3 further explanation*: A scenario cannot be completely validated until it is presented to the user. When designers create a scenario, stakeholders should validate that the scenario is correct. When a stakeholder creates or modifies a scenario, the designer must validate it. There is a subtle relationship between a 'valid' scenario and a 'validated' scenario. A validated scenario has the 'stamp of approval' from a stakeholder that it is valid. A valid scenario is a scenario that is correct (it correctly models stakeholders beliefs of how the future system should behave and the data is correct). It is unknown for sure if a scenario is valid until it is validated. A scenario, prior to being validated by the stakeholders, may already be valid or correct. Also, a scenario, prior to being validated by the stakeholders may be invalid and modified to be valid after presented to the stakeholders. See examples below for scenarios valid prior to presented to stakeholders and invalid prior to presented to stakeholders.
- ❖ *16.4 tolerance for exclusion*: MEDIUM. A scenario cannot be validated until presented to the stakeholders. This means the scenario cannot be validated at

creation. However, to help ensure the requirements team is on the right path, the sooner the scenario is validated the better.

- *16.5 justification for inclusion*: A goal of an SRS, the eventual end product of scenarios, is to be correct. One does not want to build a system that is not correct because it will not meet the stakeholders needs. Validated scenarios help ensure correctness.

❖ *16.6 Examples* (domain – Automated Teller Machine [ATM]):

Not valid prior to being presented to user	Valid prior to being presented to user
EXAMPLE: • System displays main menu • Customer presses 'withdrawal' button • System displays prompt requesting PIN • Customer enters PIN • System verifies PIN • System prompts user for amount to withdraw VALIDATION: Fail – PIN should be entered before main menu is displayed	EXAMPLE: • System displays prompt requesting PIN • Customer enters PIN • System verifies PIN • System displays main menu • Customer presses 'withdrawal' button • System prompts customer for amount to withdraw VALIDATION: Pass – 8/21/97 by Jay Billups
EXPLANATION: This portion of an ATM scenario does not correctly model the stakeholders' beliefs on the sequence of events for a user to withdraw money, PIN should be entered earlier – this scenario is not valid prior to being presented to the user	EXPLANATION: This portion of an ATM scenario does correctly model the stakeholders' beliefs of the sequence of events for a user to withdraw money – this scenario is valid prior to being presented to the user

BOTH SCENARIOS BELOW ARE THE SAME (ONE VALIDATED, ONE IS NOT):

Attribute not present (yet scenario is valid)	Attribute present
EXAMPLE: • System displays prompt requesting PIN • Customer enters PIN • System verifies PIN • System displays main menu • Customer presses 'deposit' button • System prompts customer for amount • Customer enters amount	EXAMPLE: • System displays prompt requesting PIN • Customer enters PIN • System verifies PIN • System displays main menu • Customer presses 'deposit' button • System prompts customer for amount • Customer enters amount VALIDATION: Pass – 8/21/97 by Jay Billups
EXPLANATION: This portion of an ATM scenario does correctly model the stakeholders' beliefs of the sequence of events for a user to deposit money – it happens to be valid, but the validity is unknown since it has not been validated yet.	EXPLANATION: This portion of an ATM scenario does correctly model the user's beliefs of the sequence of events for a user to deposit money – it happens to be valid which is verified upon validation

❖ *16.7 applicable to*: ☐ set of scenarios ☒ single scenario

17. Single Threaded

❖ *17.1 attribute definition:* A scenario does not include any branches, condition statements (other than initial conditions) such as if-then-else, but, instead, shows a single path interaction with the system.

- *17.2 origin*: Scenario {use case} attribute 'single threaded' (by Rumbaugh 1994).
- *17.3 further explanation*: When a user is presented with a menu, or has several options that they can take at a point in time, a scenario should model only one possible option and outcome.
- *17.4 tolerance for exclusion*: LOW. There is no benefit or reason to not have scenarios be a single thread from the beginning.
- *17.5 justification for inclusion*: If branches were included in a scenario, the scenario may become very long, unmanageable and difficult to follow.

❖ *17.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: Initial Conditions: user has entered ATM card and PIN successfully • System displays main menu with options: withdraw, deposit, check balance • *If* customer presses the 'deposit' button *then* system prompts user to enter deposit amount • After entering amount, customer drops money in deposit slot • *Else, if* user presses the 'withdraw' button *then* system prompts customer to enter withdrawal amount • Customer enters amount • *If* system can verify funds *then* system gives withdrawal amount to customer • *Else, if* system cannot verify funds, *then* system displays pop-up menu stating funds cannot be verified and returns card to user • *Else, if* customer presses 'balance' button *then* system prompts user for checking or savings balance	EXAMPLE: Initial Conditions: user has entered ATM card and PIN successfully Name: Successful deposit of funds • System displays main menu with options: withdraw, deposit, check balance • Customer presses 'deposit' button • System prompts user for deposit amount • Customer enters withdrawal amount • Customer enters deposit amount • System prompts user to place deposit in slot. Initial Conditions: user has entered ATM card and PIN successfully Name: Successful withdrawal of funds • System displays main menu with options: withdraw, deposit, check balance • Customer presses 'withdraw' button • System prompts user for withdrawal amount • System verifies funds • System gives customer money

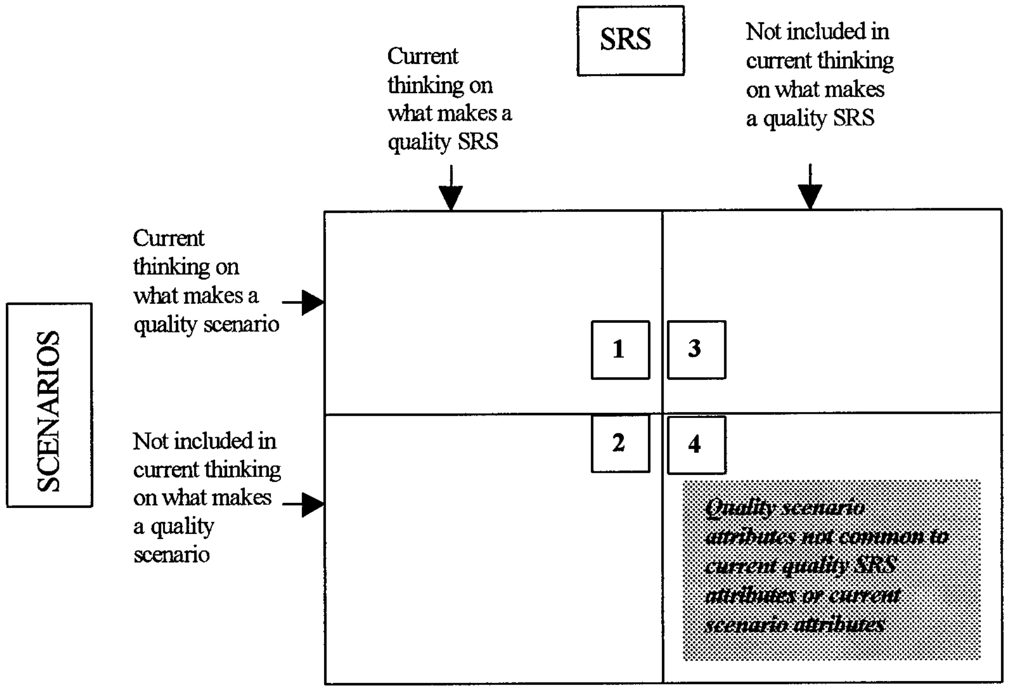
• *If* customer chooses 'savings' *then* savings balance is displayed, otherwise *if* user chooses 'checking' *then* checking balance is displayed	Initial Conditions: user has entered ATM card and PIN successfully Name: Unsuccessful withdrawal of funds (funds cannot be verified) • System displays main menu with options: withdraw, deposit, check balance • Customer presses 'withdraw' button • System prompts customer for withdrawal amount • Customer enters withdrawal amount • System cannot verify funds • System displays pop-up display stating that funds cannot be verified and cannot withdraw money Initial Conditions: user has entered ATM card and PIN successfully Name: Successful checking of savings account balance • System displays main menu with options: withdraw, deposit, check balance • Customer presses 'check balance' button • System prompts customer for 'checking'' or 'savings' balance • Customer selects 'savings' balance • System gets balance amount • System displays balance
EXPLANATION: There are multiple branches and 'if-then' conditions, making this scenario very confusing to follow.	EXPLANATION: Each path through the system is broken into its own scenario, making each scenario easier to follow and comprehend the goal being reached.

❖ *17.7 applicable to*: ☐ set of scenarios [✓ single scenario]

Quality Scenario Attributes with no Origins

Several attributes, inspired or expanded upon from previous work, have been defined. There are still more attributes that make up a quality scenario/scenario set that have not been defined yet. These attributes represent quadrant 4 as highlighted in Figure 23. From the literature search I conducted and from listening to experts in the field talk about scenarios and their uses, I have realized that there are still some attributes missing. These attributes include: modeling both normal and exceptional cases, simulating system failure and recovery, using multiple forms of media as needed (not tied to one form of media), and clearly showing the boundary between the system and the user.

Figure 23 New Attributes: Not Found in Current Thinking on Quality in a Scenario or Quality in an SRS


SRS
Current thinking on what makes a quality SRS
Not included in current thinking on what makes a quality SRS
SCENARIOS
Current thinking on what makes a quality scenario
Not included in current thinking on what makes a quality scenario
1
2
3
4
Quality scenario attributes not common to current quality SRS attributes or current scenario attributes

18. Model both Normal and Exceptional Cases

- *18.1 attribute definition:* A set of scenarios should model both normal and exceptional system interactions.
- *18.2 origin*: new
- *18.3 further explanation*: While it is important for scenarios to model normal processes and interactions, it is equally important to capture the exceptional or abnormal cases. For example, most user inputs have a range of legal values. What happens if the user attempts to enter a value outside that range? There are other cases that lie outside of user input such as an ATM machine not having

enough money to cover a withdrawal. The exceptional cases will depend on the domain being modeled.

- *18.4 tolerance for exclusion*: HIGH. When trying to determine the requirements of a system, the first step should be to model the normal activities. After they are understood, then the exceptional cases can be modeled (although they may pop up even when discussing normal cases).
- *18.5 justification for inclusion*: Understanding and agreeing upon what will happen outside input boundary situations and other exceptional cases before implementation decreases the likelihood of discrepancies between system performance and stakeholder needs. By using scenarios to model these situations bring these issues to the forefront for discussion.

❖ *18.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
	EXAMPLE: • User prompted for amount of withdrawal displaying that the amount must be a multiple of $20 • User enters a multiple other than $20 (such as $65) • System displays a pop-up menu with an 'OK' button reminding them that amount must be a multiple of 20 • User touches the ok button • User prompted for amount of withdrawal
EXPLANATION: The ATM machine being modeled only contains twenty-dollar bills. When users want to withdraw, they are directed to only withdraw multiples of $20. The set of scenarios does not capture what happens when the user requests something other than a $20 multiple.	EXPLANATION: The ATM machine being modeled only contains twenty-dollar bills. When users want to withdraw, they are directed to only withdraw multiples of $20. Included in the set of scenarios is what happens when a user enters a value other than a multiple of $20.

❖ *18.7 applicable to*: [✓ set of scenarios] ☐ single scenario

19. Simulate System Failures and Recoveries as Possible

- ❖ *19.1 attribute definition:* Known system failures (both internal to the system and external with other entities the system depends on) and the recovery from the failures should be modeled in a set of scenarios.
- ❖ *19.2 origin*: new
- ❖ *19.3 further explanation*: This is a tricky attribute because developers do not deliberately build a system to fail, nor will they know all the ways the system will crash until it is built. However, there are some possible scenarios that may cause problems for a software system such as: database full, power failure, failure to receive necessary messages from an independent system, and other possible hardware failures. In addition, some software is built with the sole purpose of watching hardware and detecting and recovering from hardware failures. Software scenarios should exist showing what happens when hardware fails. This attribute can depend greatly on the hardware suite chosen. Stakeholders do not like to think of their new system failing, and it is possible that they may get scared or lose confidence in the future system by talking about failures. However, if it can be shown that there is a graceful recovery, this can be reassuring. This attribute should be considered carefully and used subjectively and judiciously.
- ❖ *19.4 tolerance for exclusion*: LOW to HIGH (application dependent). LOW: Software systems that have the sole purpose of monitoring hardware (e.g. software program to detect if any space shuttle hardware fails) should include this attribute from the beginning. HIGH: For other programs, this attribute can wait and be added as a refining step at the bottom of the requirements funnel.

❖ *19.5 justification for inclusion*: System failure is nearly inevitable. The more customers understand what will happen in different situations, the less likely they will be shocked or surprised during testing and fielding of the system.

❖ *19.6 Example* (domain – Air warning system)

Attribute not present	Attribute present
	EXAMPLE: Initial Conditions: • Receiving aircraft update message on heading and speed Scenario: • System stops receiving aircraft update messages • System sends request for update message to radar site after waiting 5 minutes from last message • After waiting 5 minutes for response from radar site, no response it given • Radar site is marked as degraded • System dead reckons the aircraft (estimates current heading and speed after last known location)
EXPLANATION: An air warning system that relies on receiving external aircraft messages to know where each aircraft is located does not include in its set of scenarios what would occur if the external system that provides the aircraft messages crashes.	EXPLANATION: Included in a set of scenarios for an air warning system is the situation of not receiving the external aircraft messages.

- ❖ *19.7 applicable to*: [✓ ☒] set of scenarios ☐ single scenario

20. Multiple Forms of Media Used as Needed

- ❖ *20.1 attribute definition: :* A scenario should not be tied to one form of medium.
- ❖ *20.2 origin*: new
- ❖ *20.3 further explanation*: Scenario creators should use whatever media is appropriate for the scenario and mix the different types of media as needed. In the example below, the designers may have decided that a buzzer is the best way to alert a customer that their card is available, but the stakeholders prefer a verbal message. By sounding the buzzer, the stakeholders can tell the designers that they do not want the buzzer and other options can be discussed. Also, a foam mockup of the display or keyboard could supplement the scenario.
- ❖ *20.4 tolerance for exclusion*: MEDIUM. At the top of the funnel the idea is to capture ideas in whatever form is convenient. After the initial ideas are captured, other forms of media need to be incorporated to flush out the scenario and represent the scenario to a fuller capacity.
- ❖ *20.5 justification for inclusion*: Not every aspect of a scenario can be shown solely through text, a storyboard, etc. Using multiple forms of media (especially sound) appeal to the different senses, giving stakeholders a fuller understanding and comprehension of the scenario.
- ❖ *20.6 Example* (domain – Automated Teller Machine [ATM]):

Attribute not present	Attribute present
EXAMPLE: NAME: Transaction completed • Customer receives cash • System prints out receipt • Customer takes receipt • System outputs ATM card to customer • The system alerts the customer to take their card	EXAMPLE: NAME: Transaction completed • Customer receives cash • System prints out receipt • Customer takes receipt • System outputs ATM card to customer • A buzzer {sound buzzer} alerts customer to take their card
EXPLANATION: The scenario does not show or let stakeholders experience how the system alerts the customer to take their card	EXPLANATION: The scenario describes the use of a buzzer to alert the customer along with the buzzer sounding to let stakeholders hear how the alerting will happen

❖ *20.7 applicable to*: ☐ set of scenarios ☒ single scenario

21. Boundaries Between System and Users Shown Clearly

❖ *21.1 attribute definition:* A scenario should distinctly show the boundary of responsibilities between the system and the user.

❖ *21.2 origin*: new

❖ *21.3 further explanation*: There should be no confusion between the roles and responsibilities of the user and of the future system. A phrase like 'the card is given' in an ATM scenario does not say who or what is giving the card.

Stakeholders may assume differently who is performing which action. Instead, the phrase should be 'the user inserts the card' or ' the system outputs the card'.

- ❖ *21.4 tolerance for exclusion*: MEDIUM. From the beginning, the distinction should be made with respect to 'who does what?' In some cases this may not be known right away and will be discovered further down the requirements funnel. However, this should not be delayed too long, otherwise assumptions may be made regarding boundaries.
- ❖ *21.5 justification for inclusion*: It is better to get out in the open any assumptions one may have about roles and boundaries of the system and the user. If assumptions went untested until implementation and the assumptions were incorrect, it will be more expensive to fix than if discovered earlier.
- ❖ *21.6 Example* (domain – Cashier register system):

Attribute not present	Attribute present
EXAMPLE: • Cashier enters amount of last item into the register • Cashier presses the 'total' button • Total amount is displayed to the cashier on the screen	EXAMPLE: • Cashier enters amount of last item into the register • Cashier presses the 'total' button • System calculates tax on total amount • Total amount (including tax) is displayed to the cashier on the screen
EXPLANATION: Responsibility for calculating tax is not clear (is it in the total, or does the cashier have to calculate?).	EXPLANATION: Responsibility for calculating tax is clearly shown to belong to the system.

❖ *21.7 applicable to*: ☐ set of scenarios [✓ single scenario]

Uniqueness and Necessity of Five Similar Attributes: (concise, discrete, single threaded, appropriate, and right level of detail)

Five of the attributes defined in this chapter are similar. They are: concise (page 51), discrete (page 53), single threaded (page 82), appropriate (page 67) and right level of detail (page 60). Their definitions have subtle differences that may be difficult to discern. To show that each of these attributes is unique, distinct, and necessary for scenarios, I will provide five examples of scenarios. Each example possesses four of the five above-mentioned attributes. I will show that although four of the five attributes are present, the scenario is not of quality and that including all five attributes will improve the scenario. These scenario examples are not very long, therefore it may seem tolerable for an attribute to be missing in the example. Nonetheless, the examples are representative of the problems that can occur from not including an attribute.

By providing these examples I will show that the five similar attributes are indeed different and necessary. Each attribute is necessary, because, without it, the scenario is not of quality. Each attribute is different because there are no two examples that are alike. For the purpose of the 'level of detail' attribute, the examples will be written as if the project is near the bottom of the requirements funnel.

EXAMPLE: Not concise

Name: Successful access to main menu

Initial conditions: System is in ready mode and displaying welcome menu

- After the customer approaches the ATM the customer enters their ATM card
- The customer sees a prompt that is displayed by the system requesting the customer enter their PIN and it takes no longer than 5 seconds for them to see the prompt from when they enter their ATM card
- The customer, knowing their PIN, enters their PIN to the system
- Verification of the customer's PIN is done by the system and it takes no longer than 10 seconds from when the customer entered their PIN for the PIN to be verified through electronic messages with the customer's home bank
- The customer sees the main menu of options displayed by the system on the monitor no longer than 1 second from when the customer's PIN was verified

This scenario is discrete, appropriate, single threaded, and at the right level of detail but it is not concise. This is not a quality scenario because the wording of the scenario is not succinct and to the point. It is too wordy and the reader can be confused when trying to decipher what each step is trying to convey.

EXAMPLE: Not Discrete

Initial conditions: System is in ready mode and displaying welcome menu

- Customer enters ATM card
- Within 5 seconds, system prompts customer for PIN
- Customer enters PIN
- Within 10 seconds, system verifies PIN
- Within 1 second, system displays main menu of options
- Customer presses 'withdrawal' button
- System prompts user to enter amount
- User enters amount
- System verifies funds
- System disperses cash
- System prompts user asking if they want another transaction
- User presses 'yes' button
- System displays main menu of options
- User presses 'check balance' button
- System prompts user for 'checking' or 'savings' accounts
- User presses 'checking' button
- System displays checking account balance
- System prompts user asking if they want another transaction
- User presses 'no' button

This scenario is concise, appropriate, single threaded and at the right level of detail, but it is not discrete. This is not a quality scenario because there is no singular goal it is trying to accomplish, it is long and can result in 'losing' the user.

EXAMPLE: Not Appropriate

Name: Successful access to main menu

Initial conditions: System is in ready mode and displaying welcome menu

- *Customer gets off at nearest bus stop*
- *Customer pays bus driver*
- *Customer walks to ATM*
- Customer enters ATM card
- Within 5 seconds, system prompts customer for PIN
- Customer enters PIN
- Within 10 seconds, system verifies PIN
- Within 1 second, system displays main menu of options

This scenario is concise, discrete, single threaded, and at the right level of detail, but it is not appropriate. This is not a quality scenario because it is not focused on what is germane to the system (how the customer gets to the ATM has no bearing on the future system). It provides unneeded information and clouds the idea of what is important for the future system.

EXAMPLE: Not Single-Threaded

Initial conditions: System is in ready mode and displaying welcome menu

- Customer enters ATM card
- *If* card is entered in proper direction *then* system prompts user for PIN within 5 seconds of receiving card
- Customer enters PIN
- *If* system verifies PIN *then* system displays main menu of options
- *Else*, *if* system cannot verify PIN, *then* a pop-up menu with 'OK' button displays error message and prompts user for PIN
- Customer touches 'OK' button and enters PIN
- *Else*, *if* card is not entered in proper direction *then* card is returned and system displays message requesting customer re-enter card

This scenario is discrete, appropriate, concise and at the right level of detail, but it is not single-threaded. This is not a quality scenario because the 'if-then-else' branches in this scenario makes it awkward and difficult to follow.

EXAMPLE: Not Right Level of Detail

Name: Successful access to main menu

Initial conditions: System is in ready mode and displaying welcome menu

- Customer sees PIN prompt
- Customer enters PIN
- Customer sees main menu

This scenario is concise, discrete, appropriate, and single-threaded, but is not at the right level of detail. This is not a quality scenario because it does not contain enough detail. For example, how does the user get to the point of seeing the prompt for their PIN from the welcome menu? How long does it take to see the main menu? Is their ATM card involved? There are too many unanswered questions for a scenario that should be at the bottom of the requirements funnel, more detail is needed.

If any of the five attributes discussed in this section are missing, then the scenario is degraded and does not reach its full potential for requirements elicitation. The following scenario shows what a scenario with all five attributes looks like.

EXAMPLE: Concise, Discrete, Appropriate, Single-Threaded and at Right Level of Detail

Name: Successful access to main menu

Initial conditions: System is in ready mode and displaying welcome menu

- Customer enters ATM card
- Within 5 seconds, system prompts customer for PIN
- Customer enters PIN
- Within 10 seconds, system verifies PIN
- Within 1 second, system displays main menu of options

An Example of a Quality Scenario

The above scenario is concise, discrete, appropriate, single-threaded and at the right level of detail. Of all the attributes for a single scenario defined in this chapter, this scenario is missing the attributes 'traced' and 'validated'. The scenario below includes these attributes and is an example of a quality scenario for software requirements elicitation.

EXAMPLE: A quality scenario

Background information:
Creator: Kim Braun

Participating stakeholders: Alyssa Holt, Diana Spencer

Date Presented: Sept 10, 1997

Change History: Aug 12 1997: Alyssa wanted to tighten time to gain access to prompt for PIN from 10 seconds to 5 seconds

Relationship with other scenarios: must be performed before any scenario dealing with withdrawals, deposits or checking balances

Rationale for scenario: show how customer gains access to system

Criticality: HIGH

Scenario Requirement: Requirement 1.2

Other issues: Diana might like to see time constraint to verify PIN dropped to 5 seconds

Name: Successful access to main menu

Initial conditions: System is in ready mode and displaying welcome menu

- Customer enters ATM card
- Within 5 seconds, system prompts customer for PIN
- Customer enters PIN
- Within 10 seconds, system verifies PIN
- Within 1 second, system displays main menu of options

VALIDATED: PASS – by Debbie Smith September 29, 1997

Twenty-one attributes for a quality scenario/scenario set have been defined and discussed. They are inspired from previous work done on scenarios, work done in the area of quality in an SRS and from original ideas. The next chapter will validate and demonstrate necessity of each attribute.

CHAPTER 4

VALIDATION AND CONCLUSION

Chapter 4 defined 21 attributes of a quality scenario / scenario set. Each attribute has been given a justification for inclusion stating why the attribute is needed. This chapter will go one step further and validate that each attribute is necessary for a scenario / scenario set. This will be accomplished by showing how the software lifecycle will be adversely affected if an attribute is not present in a scenario / scenario set. The software lifecycle in question is the waterfall lifecycle as shown in Figure 24 (adapted from Royce 1970). In addition, the functions of Management and Configuration Management (CM) will be included to show how they, too, may be affected by missing attributes.

Figure 24 Adapted Software Waterfall Lifecycle

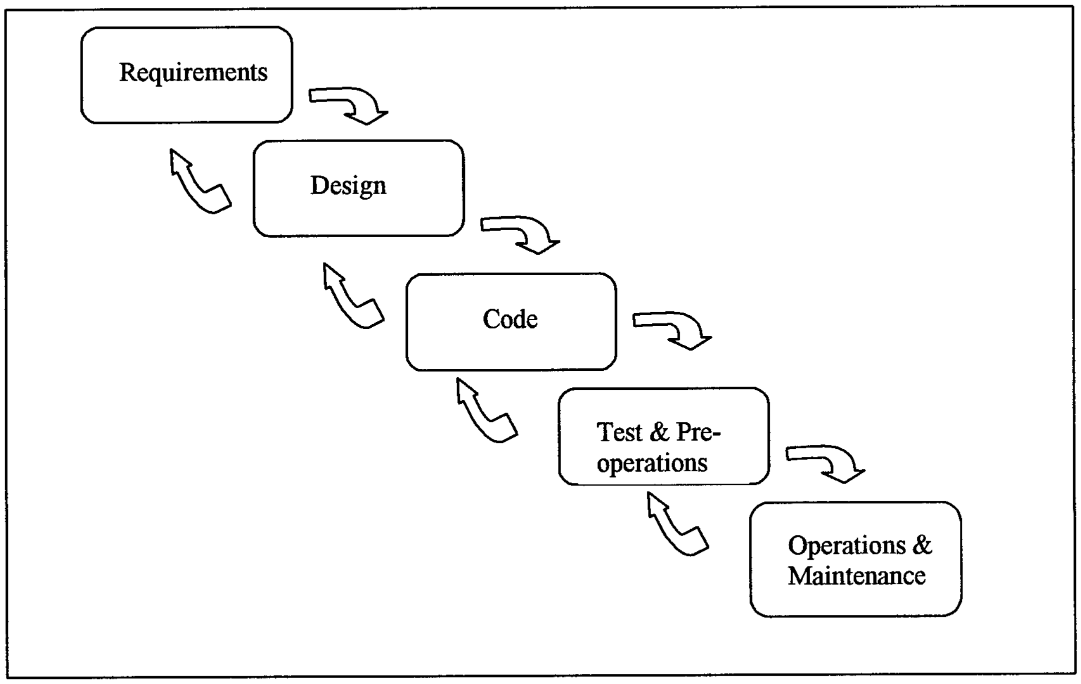


Table 5 shows the lifecycle phases along with management and CM as column headings and the scenario attributes as row headings. There are many X's in the table at column and row intersections. Each X annotates the part of the software lifecycle represented by the column it is in that will be affected if the attribute represented in that row is not present. Each row has at least one X showing that each attribute affects at least one phase of the lifecycle (including management and CM), demonstrating that each attribute is in fact necessary. The following sections of this chapter will explain how each part of the lifecycle is affected by missing attributes.

Table 5 If Scenario / Scenario Set Attribute Missing - Resulting Impact on Software Lifecycle

	REQUIREMENTS				DESIGN AND CODE				QA / TEST				Maint	Mgmt	CM
	Incomplete SRS	Incorrect SRS	SRS difficult to read	Other prblms contribut ing to bad SRS	Bad Design and Code	Diffi-cult to create design	Cost & Time	Can't use Jacobsons object model for use cases	Don't know how to test	Incorrectly pass/fail test cases	Diffi-cult to create test cases	Other prblms in QA/test	Fix system that doesn't meet needs	Mis-match between estimate and actual cost / schedule	Can't put scen-ario under CM
Complete scenario	X	X			X					X			X		
Complete scenario set	X	X			X					X			X		
Concise			X			X					X			X	
Discrete				X	X							X		X	
Single scenario consist ency		X		X	X				X				X		
Scenario set consist ency		X		X	X				X				X		
Timing constraint modeled	X				X							X	X		
Right level of detail				X	X					X			X		
Under standable		X	X			X							X		
Achiev able				X			X					X		X	
Appro priate			X			X					X	X		X	
Usable after SRS								X	X						

	REQUIREMENTS				DESIGN AND CODE				QA / TEST				Maint	Mgmt	CM
	Incomplete SRS	Incorrect SRS	SRS difficult to read	Other prblms contribut ing to bad SRS	Bad Design and Code	Diffi- cult to create design	Cost & Time	Can't use Jacobsons object model for use cases	Don't know how to test	Incorrectly pass/fail test cases	Diffi- cult to create test cases	Other prblms in QA/test	Fix system that doesn't meet needs	Mis- match between estimate and actual cost / schedule	Can't put scen- ario under CM
Named				X											X
Traced				X											
Initial conditions		X			X					X			X		
Validated													X		
Model normal and Excep- tional cases	X				X							X	X		
Simulate system failures	X				X							X	X		
Multi- media	X				X								X		
Boundries between user and system		X			X				X				X		
Single threaded			X			X					X			X	

Table 5 (Continued)

Requirements

During the requirements phase requirements are elicited from stakeholders and are specified in a document such as the Software Requirements Specification (SRS). Quality attributes missing form a scenario or scenario set can adversely affect the requirements phase by causing the SRS to be incomplete, incorrect or difficult to read. There are also other problems missing attributes can cause with respect to requirements and they will be discussed in this section as well.

Incomplete SRS

A complete SRS is one that 'exhausts all known needs and objectives' (Farbey 1990). An incomplete SRS can result if any of the following attributes are not included in a scenario or scenario set: complete (single and set), timing constraints captured, model both normal and exceptional cases, simulate system failures and recovery as possible, and use multiple forms of media as needed.

Complete scenario and complete scenario set not present. A scenario that is not complete leaves gaps such as "what is the input?" "who or what is providing the input?" "what is the output?" or "what is the system doing?". A scenario set that is not complete also leaves gaps and does not cover all actors' interactions with the system. These holes or gaps in a scenario or scenario set cause the SRS to be incomplete as they represent needs and objectives not being covered.

Timing constraints not included. When timing constraints are not covered in a scenario they are not likely to be incorporated into the SRS. This leaves the SRS incomplete since the needed timing constraints are not included.

Normal and exceptional cases not modeled; System failure and recovery not modeled. A set of scenarios that does not model exceptional cases or does not simulate system failure and recovery in its collection is less likely to produce an SRS with these cases described. This leaves the SRS incomplete because the need to handle exceptional cases and system failure and recovery is not captured.

Multiple forms of media not used as needed. Many software systems appeal to more than one of the five senses meaning multiple forms of media (such as written and sound) are needed to capture the requirements of the system completely. If a scenario does not use multiple forms of media as needed to represent how the future system will operate, then there is an 'unknown' factor as far as how the future system will work when it comes to the other forms of media not used in the scenario. This means the SRS, normally in written form, will not reference another medium to flush out the requirement(s). This leaves the SRS incomplete since the need for sound, movement, tactual, smell, etc. is not captured fully.

Incorrect SRS

In a correct SRS, 'every requirement represents something required of the system to be built' (A. Davis and Overmyer et al. 1993). An incorrect SRS can result if any of the following quality attributes are missing from a scenario or scenario set: complete (single and set), consistency (single and set), understandable, and boundaries between system and user shown clearly.

Complete scenario and complete scenario set not present. Incomplete scenarios and scenario sets leave gaps or holes. SRS writers may unknowingly fill in these gaps incorrectly when writing their document, otherwise the gaps may remain, leaving the document incomplete as explained on page 108. These incorrectly filled in gaps create incorrect requirements leaving the SRS itself incorrect.

Consistent scenario and consistent scenario set not present. An inconsistent scenario or scenario set contains contradictions. The SRS writers may incorrectly choose one of the inconsistencies to put in the SRS, leaving the SRS itself incorrect.

Not understandable. If a scenario is not understandable by the SRS writers then the scenario can be incorrectly interpreted by the writers, resulting in a requirement that is wrong. This causes the SRS to be incorrect.

Boundaries between user and system not present. If the boundaries between the user and the system are not clearly shown in a scenario then SRS writers may make incorrect assumptions regarding responsibilities. These incorrect assumptions lead to incorrect requirements resulting in an incorrect SRS.

Initial conditions not described. A scenario with no initial conditions described, yet represents a situation needing initial conditions, is likely to yield a requirement or requirements without any conditions. This incorrectly means the requirements should hold true under all conditions, yielding an incorrect SRS.

All of the above missing requirements contribute to an incorrect SRS. They yield requirements that do not represent something required of the future system to be built.

SRS Difficult to Read

'Readable' is a desired attribute of an SRS according to Farbey (1990). An SRS that is difficult to read may result if any of the following attributes are not included in a scenario: concise, appropriate, understandable and single threaded.

Not concise. A scenario that is not concise makes for a difficult time writing the SRS since there is a lot of noise and unneeded information contained in the scenario that needs to be sifted through. This can result in an SRS that is cumbersome and difficult to read because a lot of noise from the non-concise scenario may remain in the SRS.

Not appropriate. A scenario that is not appropriate contains information not germane to the future system. In turn, the SRS may contain information that is not appropriate, clouding the true requirements in the SRS. Due to the information overload with information not pertinent to the future system, the SRS becomes difficult to read.

Not Understandable. A scenario that is not understandable by the requirements writers can cause difficulties when trying to write the SRS. The SRS, or at least a portion, may be difficult to read because the scenario it originated from was hard to understand.

Not single-threaded. A scenario that is not single threaded is itself confusing and difficult to read because of all the conditional statements and branches. This can cause the SRS to be confusing, cumbersome and hard to read.

Other Problems Contributing to a Bad SRS

Other properties that cause an SRS to not be of quality include: an inconsistent SRS, an ambiguous SRS, an unachievable SRS and a non-traced SRS. In addition, when it is difficult to discern individual requirements from a scenario it can be hard to write an SRS. These problems can be caused when a scenario or scenario set does not contain the attributes: consistent (single and set), right level of detail, achievable, named, traced and discrete.

Consistent scenario and consistent scenario set not present. An internally consistent SRS is one such that 'no subset of individual requirements stated therein conflict' (A. Davis and Overmyer et al. 1993). A scenario or scenario set that is inconsistent can cause SRS writers to write an SRS that is inconsistent, meaning individual requirements conflict. Otherwise, if one of the inconsistencies is chosen to be included in the SRS at the exclusion of the other(s), then an incorrect SRS may result as described on page 110.

Not at right level of detail. By the time the SRS is written, the requirements team is at the bottom of the requirements funnel (page 40). A scenario should be more detailed than abstract. Scenarios that are too abstract leave many ambiguities and unanswered questions and lead to an SRS that is ambiguous.

Not achievable. An achievable SRS is one such that 'there could exist at least one system design and implementation that correctly implements all requirements stated in the SRS' (A. Davis and Overmyer et al. 1993). If a scenario is not achievable then the SRS written from that scenario will contain at least one requirement, and consequently an SRS, which is not achievable.

Not named or traced. A traced SRS is one such that 'the origin of each requirement is clear' (A. Davis and Overmyer et al. 1993). Therefore, it is important for each requirement in the SRS to state which scenarios it originated from. If a scenario is not named then it is not possible for a requirement to reference its' scenario origin. In addition, if a scenario is not traced, meaning the history and origin behind the scenario is not included, then a requirement in the SRS cannot trace all the way back to its beginnings (such as why the scenario was created, who created it, etc.). The result could be that changes in the SRS will have unknown operational impact.

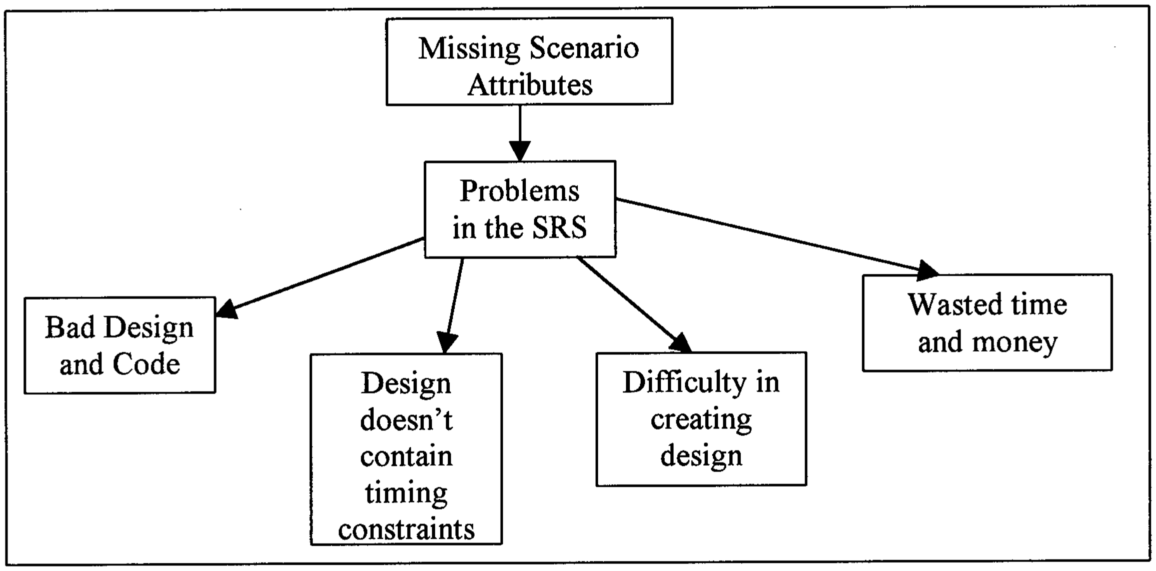
Not discrete. A scenario that is not discrete accomplishes multiple goals and can be difficult to decipher which sequence of actions or steps in a scenario can stand by themselves. This causes difficulties in deciphering individual requirements for the SRS.

Design and Code

The next phase of the software lifecycle is design. The design phase is usually further broken into two phases: preliminary design and detailed design. During design, requirements in the SRS are translated into high-level preliminary software design and then to low level detailed design. After design, the next phase is the coding phase where the detailed design is translated to compilable code. For the purpose of looking at the effect of missing scenario / scenario set attributes on the lifecycle, the design and code phases will be looked at together.

Missing scenario or scenario set attributes can adversely affect design and code. Missing attributes can cause: bad design and code, a design without timing constraints, difficulty in creating a design, wasted cost and time, and inability to use Jacobson's object use cases in design. As Figure 25 shows, many of the problems in design and code from missing scenario / scenario set attributes stem from problems the missing attributes caused in the SRS.

Figure 25 Design and Code Problems Caused by Problems in the SRS Resulting From Missing Scenario / Scenario set Attributes


Missing Scenario Attributes
Problems in the SRS
Bad Design and Code
Design doesn't contain timing constraints
Difficulty in creating design
Wasted time and money

Bad Design and Code

Bad design and code can be caused if any of the following scenario / scenario set attributes are missing: complete (single and set), discrete, consistency (single and set), timing constraints modeled, right level of detail, initial conditions described, normal and exceptional cases modeled, system failure and recovery simulated as needed, and boundaries between use and system shown clearly.

If missing, the above attributes yield problems with the SRS (the driving force of design) as described starting on page 108. A bad SRS will cause problems in design and, consequently, in code.

Scenarios / scenario sets that are not: complete, consistent, at right level of detail, containing initial conditions, containing exceptional cases or system failures, using multiple forms of media, or clearly showing boundaries between user and system lead to incorrect design and code. If the above attributes are not present they lead to problems with the SRS. Many of these missing attributes leave gaps in scenarios or scenario sets that can be incorrectly filled in when the SRS is written. This causes the SRS to be incorrect and, subsequently, leads to an incorrect design. On the other hand, the gaps may remain in the SRS and be incorrectly filled in by the designers, or left as gaps in the design leading to an incorrect or incomplete design.

A scenario that is not discrete can yield a design that is not discrete. A scenario that is not discrete makes it difficult to discern individual requirements when writing the SRS. This can yield a design that is not discrete, meaning, in order to accomplish a goal, several non-necessary steps may be designed and coded into the system incorrectly.

A scenario without timing constraints modeled yields an SRS without timing constraints for the requirements that scenario represents. This leads to a designed and coded system with arbitrary (if any) timing constraints.

Difficult to Create a Design

Scenarios that are not concise, not understandable, not single-threaded or not appropriate cause the SRS to be difficult to read. An SRS that is difficult to read means it is hard to tell exactly what the SRS is conveying and will make for design and code that is difficult to create since much time will be spent on trying to decipher the SRS. This leaves room for interpretation with respect to what the design should accomplish.

Money and Time Wasted

A non-achievable SRS can increase the cost and time needed for design and code. A scenario that is not achievable creates an SRS that is not achievable. Wasted time, money and effort will go into trying to create a design and then in trying to code some aspect of the proposed system that cannot be achieved. Money will be wasted on trying to design something that cannot be created for scope or technology reasons.

Jacobson's Object Use Cases cannot be used

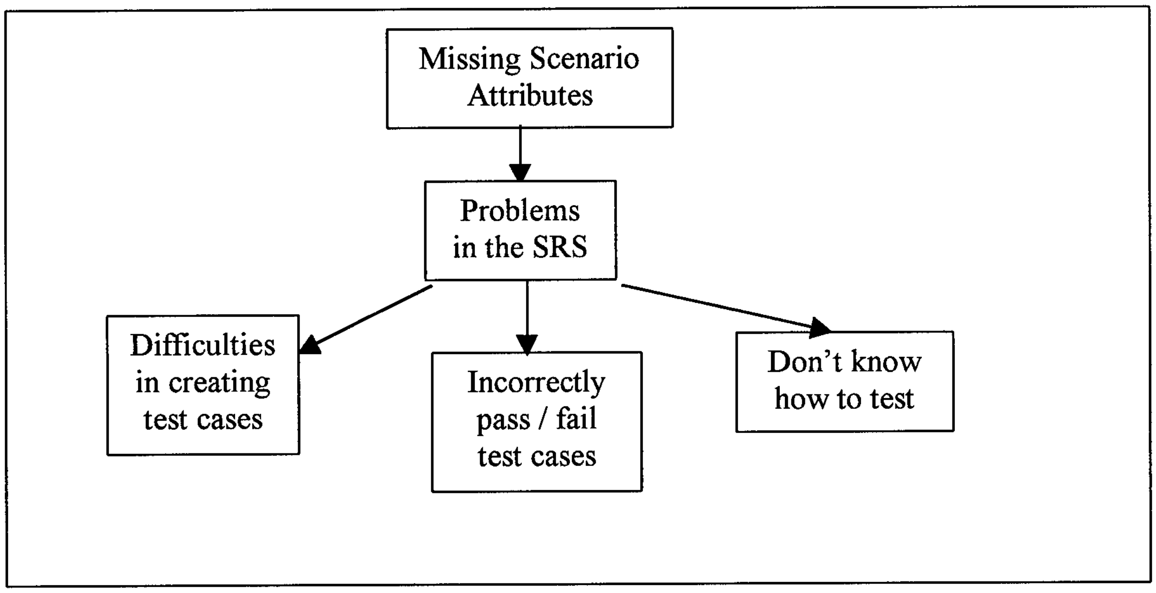
A scenario that is not usable after the SRS is written does not allow for use in design. As explained on page 12, Ivar Jacobson uses use cases for object oriented design via object use cases. A scenario that is not usable after the SRS is written does not only cause problems for testing, but gets in the way of using Jacobson's method for converting use cases used in requirements to those used in object oriented design.

Test

Following the coding phase is the test phase. Testing can be broken into: unit testing, integration testing and system testing. For the purpose of looking at the effects of missing scenario / scenario set attributes on testing, this paper will focus on system testing. System testing is where the coded, compiled and finished system is tested to see if it meets the requirements as stated in the SRS. Missing scenario / scenario set attributes can cause problems in the test phase such as: not knowing how to test a scenario, incorrectly passing/

failing test cases, difficulties in creating test cases and other problems in testing. As with the design and code phases, many of the problems in the testing phase resulting from missing scenario / scenario set attributes stem from the problems the missing attributes caused in the SRS. Figure 26 shows some of these problems stemming from a bad SRS.

Figure 26 Testing Problems Caused by Problems in the SRS Resulting from Missing Scenario / Scenario Set Attributes


Missing Scenario Attributes
Problems in the SRS
Difficulties in creating test cases
Incorrectly pass / fail test cases
Don't know how to test

Don't Know How to Test a Scenario

Scenarios or scenario sets that are not consistent, not usable after the SRS is written or without boundaries between the system and user clearly shown cause problems in knowing how to test the system. Scenarios or scenario sets that are inconsistent lead to an SRS, design, and code that are inconsistent. Testing an inconsistent system presents problems. Different sections of an inconsistent SRS may require the system, under identical circumstances, to behave differently. How can one test that a system simultaneously behaves differently or contradictory? A system that is not testable (usable after the SRS is written) means there is no way to test that the validated scenario is satisfied by the finished system or, ultimately, that the system meets the stakeholders' needs. A scenario that does not clearly show the boundaries between the user and system is likely to produce an SRS without clearly defined boundaries between the user and the system. When it comes to testing, it will be hard to know how to test an unclear boundary.

Incorrectly Pass / Fail Test Cases

Scenarios or scenario sets that are not complete, not detailed enough or without initial conditions described can cause test cases to be incorrectly passed or failed. This means test cases may be passed or failed based on a poor scenario / scenario set or SRS that does not properly reflect stakeholders' needs. The main reason for this problem is that these missing attributes leave room for interpretation by the testers.

Scenarios or scenario sets that are incomplete have holes or gaps that lead to an incomplete SRS, design, and coded system causing problems in testing. When these holes are discovered during testing there is room for interpretation on what should happen. For instance, page 50 shows an example of an incomplete scenario set. While it appears to be a blatant error, assume the SRS does not capture what happens when the user wants to check his/her balance (although it is an option on the menu). During testing, a tester selects 'check balance'. How do testers know if what happens next is correct and therefore should pass the test, or is wrong and should fail the test? The tester may make incorrect assumptions and incorrectly pass or fail the test case.

Scenarios that are not detailed enough are vague and ambiguous leading to assumptions during testing. They leave room for interpretation with respect to exactly what the finished system should do. This means the tests may be incorrectly passed or failed depending on the testers' interpretation of the scenario and how the test case was written.

If initial conditions are not described in a scenario then testers will not know if any special circumstances must be in place for the scenario to succeed. This means test cases may be erroneously passed or failed because the initial conditions are not in place.

Difficulties in Creating Test Cases

Scenarios that are not concise, appropriate or single-threaded cause difficulties when trying to create test cases. They lead to an SRS that is difficult to read. An SRS that is difficult to follow will make it difficult to create test cases from the SRS. In addition, these three missing attributes cause the scenarios themselves to be confusing, making it difficult to discern test cases straight from the scenarios.

Other Problems in Testing

Long test cases, testing for non-appropriate system aspects and waiting arbitrarily long for test results are some of the other problems missing scenario attributes can cause with respect to testing. This section will discuss these other problems.

A scenario that is not discrete, not appropriate or does not have timing constraints causes testing problems with regards to length. A scenario that is not discrete is long and creates long test cases. A scenario that is not appropriate contains information not germane to the future system meaning test cases created from the scenario or resulting SRS will test for non-appropriate aspects of the environment / system. This can lengthen the time it takes to test. A scenario without timing constraints leads to a system with arbitrary or no timing constraints. When testing, this means the tester may have to wait an arbitrarily long time to see results from their test cases.

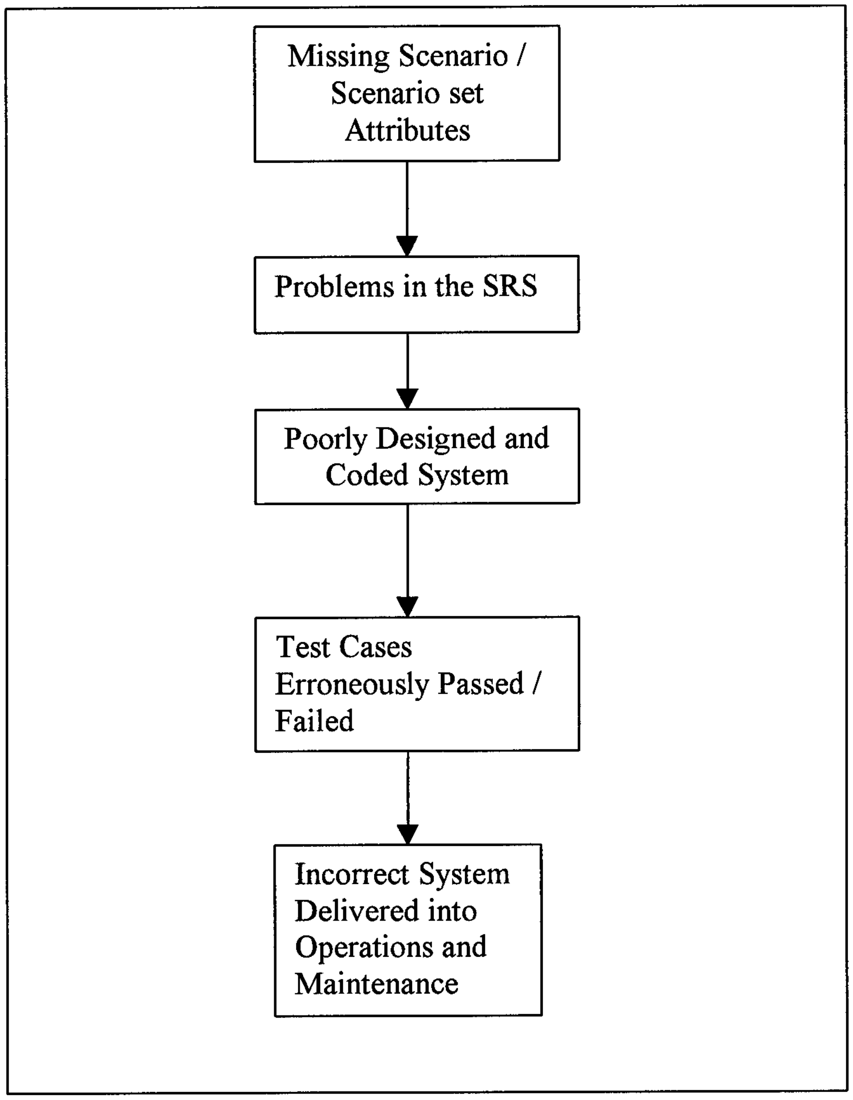
A scenario or scenario set that is not achievable, does not model exceptional cases or simulate system failures creates problems in testing. A scenario that is not achievable leads to a system that cannot be created. This means the test cases from this unachievable scenario will fail because the system cannot be built to pass the test cases. A scenario set that does not model exceptional cases or simulate system failures means the resulting system will not handle these unique situations to any specification. With respect to testing, test cases will not be created to check for the proper handling of these situations, meaning a system that does not meet stakeholder needs regarding system failures and recoveries or exceptional cases may pass testing.

Maintenance

After a system has passed testing it moves to the operations and maintenance phase until the system is retired. During maintenance, the system is *corrected* to fix any problems it was delivered with and *adapted* to meet changing needs. Between 60% - 80% of total software dollars are spent on maintenance (Pigoski 1997) making it the most expensive phase in the software lifecycle. Missing scenario or scenario set attributes can cause an incorrect system (one that does not meet stakeholders' needs) to be delivered. This requires the system be corrected during operations and maintenance. As Figure 27 shows, missing scenario / scenario set attributes cause problems in requirements, design and coding, and testing.

Ultimately, these problems cause the final system to be delivered with problems and in need of correction during maintenance.

Figure 27 Source of Problems in Maintenance Stemmed from Missing Scenario / Scenario Set Attributes


Missing Scenario / Scenario set Attributes
Problems in the SRS
Poorly Designed and Coded System
Test Cases Erroneously Passed / Failed
Incorrect System Delivered into Operations and Maintenance

Correcting the Delivered System

A scenario or scenario set that is not complete or not consistent can cause the system to be incorrect when delivered. An incomplete scenario or scenario set can cause the final system to be incomplete when delivered. This means the gaps or holes need to be filled during maintenance. An inconsistent scenario or scenario set can cause the delivered system to be incorrect and/or inconsistent and will require the system be fixed in maintenance to meet users' needs.

A scenario without timing constraints or without initial conditions described can cause an incorrect system to be delivered with maintenance required. A scenario without timing constraints yields a delivered system without regard to timing. This can create unhappy users (among other problems) because of the poor timing of the system. For example, a 'welcome and instructions' sequence of screens may flash by the user, or, it may take too long for a user to get a response from the system. It can also cause problems for other entities relying on this system for messages. A system needing a faster response time may require additional hardware causing a major and expensive system redesign in maintenance. A scenario without initial conditions described can yield an unconditional system (or at least a system with some unconditional aspects). This means the system may react in ways that the stakeholders do not desire. The result is the system will need to be fixed in maintenance.

A scenario that is not detailed enough, does not clearly show boundaries between the user and system, or is not understandable creates an incorrect system. A scenario that is not detailed enough or does not clearly delineate the boundaries between the user and the system can be vague with several different interpretations possible. This ambiguity can cause testers to pass the system although the system does not truly meet stakeholders' needs. This will cause the system to be in need of repair in maintenance. When stakeholders do not understand a scenario they may validate a scenario (page 79) that does not truly reflect how they want the system to behave because they do not understand what the scenario is doing. The system may be built and tested to meet the validated scenario even though the scenario is wrong. During operations, it will come to light that the system does not meet stakeholders' needs and will need to be corrected during maintenance.

A scenario that is not validated may not be valid or correct. This means the final system developed from a non-validated scenario may not be correct in meeting stakeholders' needs. Therefore, the system will need to be corrected in maintenance.

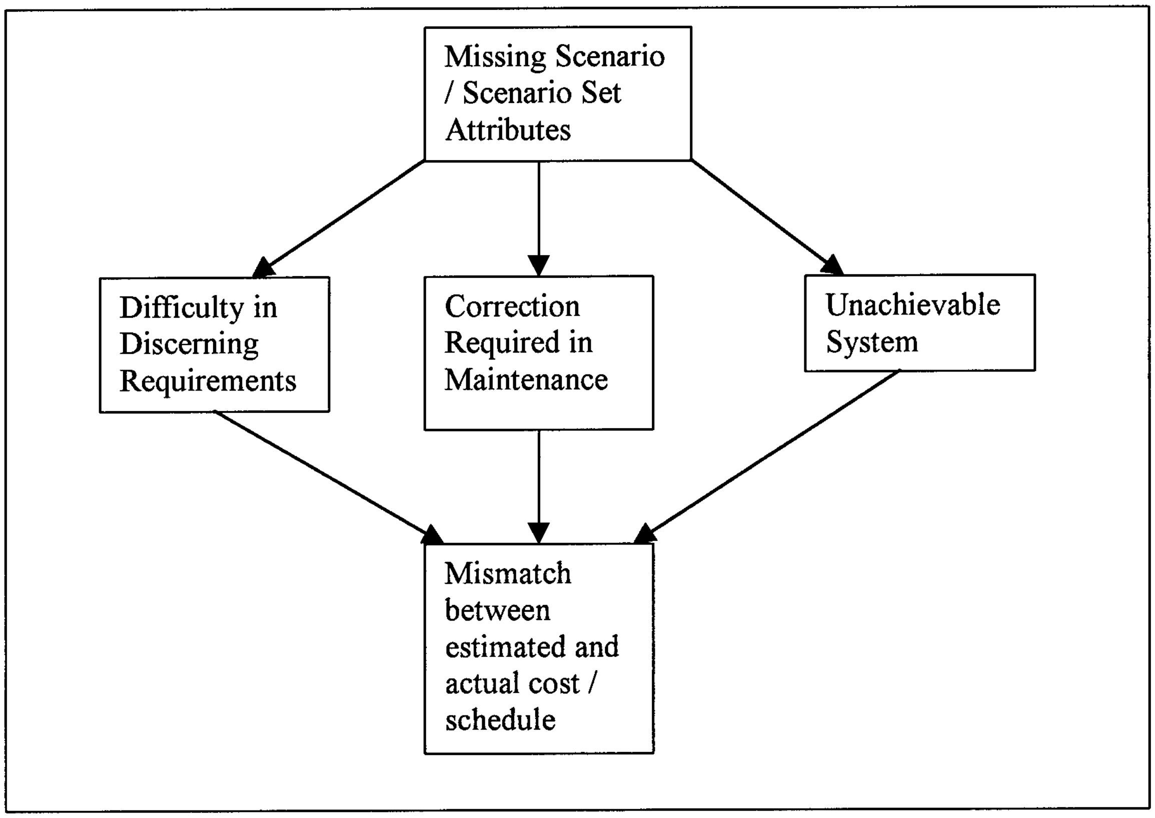
A scenario set that does not model exceptional cases or simulate system failure and recovery mean situations in the final system may develop that have unknown results. If these situations are not modeled or discussed during requirements and design then one can only guess at what will happen when an exceptional case or system failure occurs. These cases will surface once the system is operational (if not before) and the system may need to be fixed if the system does not properly handle the situation.

When multiple forms of media are not used in a scenario that describes a situation appealing to more than one of the five senses then an incomplete SRS and a system that does not meet stakeholders' expectations can be the result. For example, a system may require a verbal message to be sounded at a certain point. Designers and stakeholders may agree on the words that should be spoken via a written scenario. The designers may implement a computerized voice when the other stakeholders wanted a human, male voice. If a recording of the message were played at the time the scenario was presented, this difference in ideas would have been caught. Instead, the system will need to be corrected in maintenance.

Management

While management is not a phase in the software lifecycle, it is an important function for all software projects. Among the management duties are planning for and controlling resources. For each project, management needs to know how many people, how much time, and how much money to put on the project. They must estimate how many of these resources will be required for the project to ensure the project will be successful, yet will not waste resources. Some missing scenario attributes can cause a mismatch between management estimates and the actual use of resources such as cost and schedule. As Figure 28 illustrates, this mismatch can stem from the correction of the system required in maintenance, difficulties in discerning requirements, and an unachievable system that may be caused by missing scenario / scenario set attributes.

Figure 28 Possible Causes for Mismatches in Estimated and Actual Cost / Schedule


Missing Scenario / Scenario Set Attributes
Difficulty in Discerning Requirements
Correction Required in Maintenance
Unachievable System
Mismatch between estimated and actual cost / schedule

Mismatch Between Estimated and Actual Cost / Schedule

The previous section on problems in maintenance (beginning on page 120) discussed how if any one of nearly 15 scenario / scenario set attributes were missing they would cause the final system to be in need of correction in maintenance. The cost of putting these attributes in the scenario, ensuring that the requirements, design and code were correct would be less expensive than to fix the problems in maintenance (Boehm 1981). Management does not necessarily plan for scenarios to be incomplete, inconsistent, etc. and will not allocate funds in maintenance to correct such problems. Therefore, when the system is in need of correction in maintenance due to missing attributes, there is a mismatch between management's estimated cost and the actual cost. There are additional missing attributes that may lead to a mismatch between estimated and actual cost / schedule. They are: concise, discrete, achievable, appropriate, and single threaded.

Scenarios that are not concise, discrete, appropriate or single-threaded create a difficult time in discerning requirements. It takes people longer to try and sift through the noise, non-appropriate information and difficult to follow writing of scenarios that do not have these attributes. These missing attributes may result in an SRS that is difficult to follow, cumbersome and, consequently, require a longer time to create a design from such an SRS. Management may not have estimated the extra time needed by their people to sort through and analyze such material meaning the estimated time will be less than the actual time taken to write an SRS, design, etc. This results in a slip in the planned / estimated schedule.

A scenario that is not achievable results in an SRS, design and system that is not achievable. If it is discovered before developers try and design the system that it is unachievable then management's corresponding estimates will not be valid and the project will either be scrapped or re-evaluated with new estimates created. However, if it is not caught prior to design that the system is unachievable then an infinite amount of time and money will be spent on trying to design and create a system that cannot be created (until someone 'pulls the plug' on the project). Since management did not know the system was unachievable (otherwise the project would not have gone forward), their estimates will be less than the actual time and cost needed.

Configuration Management (CM)

While not a phase in the software lifecycle, CM is yet another important function for software projects. CM controls the current version of code and accompanying documents (also called the baseline). CM allows people to know exactly what comprises the current operational baseline, test baseline, etc. It allows people to know the contents of previous versions of the system, what changes were made, and so on. This is helpful in trying to pin down in what version bugs in the software appeared. CM also strictly controls who can change the material under its control, recording names and dates. Not only should code be under CM control, but also documentation, requirements and scenarios. It is important to have scenarios under CM control so that changes made to scenarios are controlled and documented, and previous versions can be referenced.

Inability to put Scenarios under CM Control

If a scenario is not named then there is no way to reference it, meaning the scenario cannot be put under CM control. An arbitrary numbering scheme could be created, but without a name for a scenario there is not way to match the number of the scenario to the actual scenario.

Conclusion

Scenarios are very useful in requirements elicitation by prompting the discussion and sharing of ideas among stakeholders. Scenarios can record these ideas in a variety of formats as discussed in chapter 2. This paper looked specifically at written, natural language scenarios. A quality scenario / scenario set sets the stage for the best SRS, design, code, testing and delivered system possible. This paper defined a quality scenario / scenario set as being one that contains the 21 attributes described in chapter 3. Missing attributes cause problems in the software lifecycle and other software functions as shown in the first part of this chapter.

In this thesis I contributed new information to the body of knowledge on scenarios for requirements elicitation. I expanded on scenario attributes previously referenced by other authors. I also looked to SRS attributes and adapted and expanded the appropriate attributes and adopted them for scenario attributes. Lastly, I created brand new attributes not previously referenced before in the realm of a scenario or SRS. In total, I defined 21 attributes of a quality scenario / scenario set. Previously, no more than 5 scenario attributes have been mentioned at one time (Nardi 1995). Besides defining each attribute and justifying its inclusion, I provided examples of what a scenario or scenario set would be like if the attribute was not present and examples of what the scenario or scenario set wold be like if the attribute was present.

In addition to defining more attributes in a paper than have been stated before at one time, I have demonstrated in this chapter that each attribute is necessary. I showed necessity by explaining how if any one of the attributes I defined was missing, the software lifecycle or other software functions would be adversely affected.

Since it is least expensive to fix problems in the requirements phase than in the remaining lifecycle phases (Boehm 1981), it is important to flush out and refine as many correct requirements as possible. Quality scenarios and scenario sets used in requirements elicitation build a good foundation for a robust SRS and successful software project.

REFERENCE LIST

Anderson, John, and Brian Durney. 1992. Using Scenarios in Deficiency-driven Requirements Engineering. In *IEEE International Symposium on Requirements Engineering*. Los Alamitos, California: IEEE Computer Society Press.

Andriole, Stephen. 1989. *Storyboard Prototyping*. Wellesley, Massachusetts: QED Information Systems, Inc.

Backus, J. 1959. The Syntax and Semantics of the Proposed International Algebraic Language of Zurich ACM-GAMM Conference. In *Proceedings International Conference on Information Processing*. Paris: UNESCO.

Boehm, B. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, Inc.

Campbell, Robert. 1992. Will the Real Scenario Please Stand Up?. *SIGCHI Bulletin* 24 (April):6-8.

Carroll, John. 1995. The Scenario Perspective on System Development. Ch1 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Chin, George Jr., Mary Rosson, and John Carroll. 1997. Participatory Analysis: Shared Development of Requirements from Scenarios. In *CHI 97*. Atlanta: ACM.

Dasarathy, B. 1985. Timing Constraints of Real-Time Systems: Constructs for Expressing them, Methods of Validating them. *IEEE Transactions of Software Engineering* SE-11(January):80-86.

Davis, Alan. 1990. System Testing: Implications of Requirements Specifications. *Information and Software Technology* 32(6):407-414.

Davis, Alan, and Tomlison Rauscher. 1979. Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications. Paper presented at IEEE Specifications of Reliable Software Conference.

Davis, Alan, Scott Overmyer, Kathleen Jordan, Joseph Caruso, Fatma Dandashi, Anhtuan Dinh, Gary Kincaid, Glen Ledeboer, Patricia Reynolds, Pradip Sitaram, Anh Ta, and Mary Theofanos. 1993. Identifying and Measuring Quality in a Software Requirements Specification. In *IEEE First International Software Metrics Symposium*. Los Alamitos, California: IEEE Computer Society Press.

DeMarco, Tom. 1996. Keynote Address at 2nd IEEE International Conference on Requirements Engineering. Colorado Springs, Colorado. March 1996.

Erickson, Thomas. 1995. Notes on Design Practice: Stories and Prototypes as Catalysts for Communication. Ch2 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Farbey, J. 1990. Software Quality Metrics: Considerations about Requirements and Requirements Specifications. *Information and Software Technology* 32(1):60-64.

Goguen, Joseph. 1993. Social Issues in Requirements Engineering. In *IEEE International Symposium on Requirements Engineering.* Los Alamitos, California: IEEE Computer Society Press.

Holbrook, Hilliard. 1990. A Scenario-Based Methodology for Conducting Requirements Elicitation. *ACM Sigsoft Software Engineering Notes* 15 (January):95-104.

Hooper, James, and Pei Hsia. 1982. Scenario-Based Prototyping for Requirements Engineering. *ACM Sigsoft Software Engineering Notes* 7 (December):88-91.

Hsia, Pei, Jayarajan Samuel, Jerry Gao, David Kung, Yasufumi Toyoshima, and Cris Chen. 1994. Formal Approach to Scenario Analysis. *IEEE Software* 11 (March):33-41.

Jacobson, Ivar. 1995. The Use-Case Construct in Object-Oriented Software Engineering. Ch 12 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Karat, John. 1995. Scenario Use in the Design of a Speech Recognition System. Ch 5 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Keller, Steven, Laurence Kahn, and Roger Panara. 1990. Specifying Quality Requirements with Metrics. In *IEEE Computer Society Press Tutorial*. Los Alamitos, California: IEEE Computer Society Press.

Kramer, Jeff, and NG Keng. 1988. Animation of Requirements Specifications. *Software – Practice and Experience* 18(August):112-137.

Kyng, Morten. 1992. Scenario? Guilty!. *SIGCHI Bulletin* 24 (October):8-9.

Kyng, Morten. 1995. Creating Contexts for Design. Ch 4 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Leite, Julio, Gustavo Rossi, Federico Balaguer, Vanesa Majorana, Gladys Kaplan, Graciela Hadad, and Alejandro Oliveros. 1997. Enhancing a Requirements Baseline with Scenarios. In *IEEE International Symposium on Requirements Engineering*. Los Alamitos, California: IEEE Computer Society Press.

Macaulay, Linda. 1992. Requirements Capture as a Cooperative Activity. In *IEEE International Symposium on Requirements Engineering*. Los Alamitos, California: IEEE Computer Society Press.

Muller, Michael, Leslie Tudor, Daniel Wildman, Ellen White, Robert Root, Tom Dayton, Rebecca Carr, Barbara Diekmann, and Elizabeth Dykstra-Erickson. 1995. Bifocal Tools for Scenarios and Representations in Participatory Activities with Users. Ch 6 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Nardi, Bonnie. 1995. Some Reflections on Scenarios. Ch 15 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Nielsen, Jakob. 1995. Scenarios in Discount Usability Engineering. Ch 3 *Scenario-Based Design Envisioning Work and Technology in System Development*, edited by John Carroll. New York: John Wiley & Sons, Inc.

Pigoski, Thomas. 1997. *Practical Software Maintenance*. New York: John Wiley & Sons, Inc.

Potts, Colin, Kenji Takahashi, and Annie Anton. 1994. Inquiry-Based Requirements Analysis. *IEEE Software* 11 (March):21-32.

Roman, Gruia-Catalin. 1985. A Taxonomy of Current Issues in Requirements Engineering. *IEEE Computer*. 18(April):14-21.

Rosson, Mary, and John Carroll. 1995. Narrowing the Specification-Implementation Gap in Scenario-Based Design. Ch 10 *Scenario-Based Design Envisioning Work and Technology in System Development,* edited by John Carroll. New York: John Wiley & Sons, Inc.

Royce, W. 1970. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of WESCON.* Referenced in Alan Davis, System Testing: Implications of Requirements Specifications. *Information and Software Technology* 32(6):407-414.

Rumbaugh, James. 1994. Getting Started – Using Use Cases to Capture Requirements. *Journal of Object-Oriented Programming* (September):8-12,23.

Sutcliffe, Alistair. 1997. A Technique Combination Approach to Requirements Engineering. In *IEEE International Symposium on Requirements Engineering*. Los Alamitos, California: Computer Society Press.

UCCS Center for Software Systems Engineering. 1997. Formalization of Scenarios. University of Colorado, Colorado Springs.

Wexelblat, Alan. 1987. Report on Scenario Technology. *MCC Technical Report STP-139-87*. Austin, Texas: Microelectronics and Computer Tecnology Corporation.

Wright, Peter. 1992. What's in a Scenario? *SIGCHI Bulletin* 24 (October):11-12.

Young, Richard, and Philip Barnard. 1992. Multiple Uses of Scenarios: a Reply to Campbell. *SIGCHI Bulletin* 24 (October):10.

Zahniser, Richard. 1993. Storyboarding Techniques. *American Programmer*. (September):9-14.